Vorlesung

Systemprogrammierung I

Wintersemester 2001/2002

A Organisatorisches

1 Dozent

- Dr.-Ing. Franz J. Hauck
  - Lehrstuhl für Verteilte Systeme und Betriebssysteme, Informatik 4 (Prof. Dr. F. Hofmann)
  - E-mail: hauck@informatik.uni-erlangen.de

2 Übungsbetreuung

- Dipl.-Inf. Christian Wawersich — wawersich@informatik.uni-erlangen.de
- Dipl.-Inf. Meik Felser — felser@informatik.uni-erlangen.de
- Dr.-Ing. Jürgen Kleinöder — kleinoeder@informatik.uni-erlangen.de
- Studentische Hilfskräfte

3 Studiengänge

- ★ Diplomstudiengang Informatik (3. Sem.)
- ★ Lehramtstudium Informatik
- ★ Bachelorstudiengang Computational Engineering (3. Sem.)
- ★ Diplomstudiengang Wirtschaftsinformatik (ab 5. Sem.)
- ★ Bachelorstudiengang Mathematik mit Schwerpunkt Informatik (3. Sem.)
- ★ Magisterstudiengang Linguistische Informatik (3. Sem.)
- ★ Bachelorstudiengang Linguistische Informatik (5. Sem.)

- Vorlesung und Übung
  - Anrechenbare SWS: 4 SWS Vorlesung, 4 SWS Übungen

4 Inhalt

- Vorlesung
  - Grundlagen der Betriebssysteme (eingeschränkt auf Monoprozessoren)
  - Konzepte moderner Betriebssysteme
  - Beispielhafte Betrachtung von UNIX, Linux, Windows, Windows NT/2000
- Übungen
  - Umgang mit den in der Vorlesung vorgestellten Betriebssystemkonzepten
  - Betriebssystemschnittstelle des UNIX/Linux-Betriebssystems (POSIX)
  - Umgang mit sogenannten *System-Calls*
  - Praktische Arbeiten: Ausprogrammieren von Übungsaufgaben in der Programmiersprache C

5 Vorlesung

- Termine: Mo. von 10.15 bis 11.45 im H7
  Do. von 16.15 bis 17.45 im H7
- Skript
  - zwei Alternativen:
    - Folien der Vorlesung werden im WWW zur Verfügung gestellt und können selbst ausgedruckt werden
      (Vorteil: evtl. Farbe)
    - Folien werden kopiert und vor der Vorlesung ausgegeben;
      Gutscheinverkauf, Kosten 10,00 DM für knapp 600 Folien auf ca. 150 S.
      (Vorteil: das Ausdrucken wird nicht vergessen)
  - weitergehende Informationen zum Nachlesen findet man am Besten in der angegebenen Literatur

5 Vorlesung (2)

- URL zur Vorlesung
  - http://www4.informatik.uni-erlangen.de/Lehre/WS01/V_SP1/
  - hier findet man Termine, Folien zum Ausdrucken und Zusatzinformationen

- Literatur
  - A. Silberschatz; P. B. Galvin: *Operating Systems Concepts*. 4th Edition, Addison-Wesley, 1994.
  - A. S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1992.
  - R. W. Stevens: *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

5 Vorlesung (3)

- Rückmeldungen und Fragen
  - Geben Sie mir Rückmeldungen über den Stoff. Nur so kann eine gute Vorlesung entstehen.
  - Stellen Sie Fragen!
  - Machen Sie mich auf Fehler aufmerksam!
  - Nutzen Sie außerhalb der Vorlesung die Möglichkeit, elektronische Post zu versenden: hauck@informatik.uni-erlangen.de !

6 Übungen

- Übungsbeginn ist Montag, 22. Oktober 2001

- In der Woche vom 22. bis 26. Oktober 2001
  - Übungen in Großgruppen
  - Termine
    - Di. 16.00 – 17.30 im H9
    - Do. 10.15 – 11.45 im H9
    - Do. 12.30 – 14.00 im H7

6 Übungen (2)

- Danach
  - Beginn von Übungen in Kleingruppen

- Aufteilung
  - Tafelübung – 2 SWS
  - Rechnerübung – 2 SWS

- Anmeldung zur Übung und Einteilung in die Übungsgruppen
  - „login: span“ an allen CIP-Workstations der Informatik
  - Login-Freischaltung ab noch nicht bekannt
  - Benötigte Eingaben: persönliche Daten, Matrikelnummer, Termin der gewünschten Tafelübungen
  - Jede(r) bekommt einen Übungsplatz!

6.1 Tafelübungen

- Termine

  stehen noch nicht fest

- Im Zweifelsfalls sind die Termine aus dem Anmeldeprogramms richtig

6.1 Tafelübungen (2)

- Inhalt
  - Kurzeinführung in die Programmiersprache C
  - Besprechung von Übungsaufgaben
  - Klärung offener Fragen
  - Vermittlung ergänzender Informationen zur Vorlesung

6.2 Rechnerübungen

- Termine

stehen noch nicht fest

6.2 Rechnerübungen (2)

- Inhalt
  - Lösung der Übungsaufgaben (durch die Studierenden)
  - Raum 01.155 ist reserviert
    (Vorrang am Rechner für Übungsteilnehmer)
  - ein Übungsleiter steht für Fragen zur Verfügung

7 Studien- bzw. Prüfungsleistungen

- keine
  - Informatik Lehramt
- Schein
  - Informatik (Diplom)
  - Mathematik mit Schwerpunkt Informatik (Bachelor)
  - Linguistische Informatik (Magister, Bachelor)
- studienbegleitende Prüfung
  - Computational Engineering (Bachelor)
  - Wirtschaftsinformatik (Diplom)

7.1 Schein

- ★ Verpflichtende Abgabe von Übungsaufgaben
  - Abgabe erfolgt über ein spezielles Abgabeprogramm
  - Aufgaben werden auf Plausibilität geprüft und auf Abschreiben getestet
  - Aufgaben werden in Stichproben genauer analysiert
- ★ Klausur am Semesterende (60 min)
  - Zulassung zur Klausur nur wenn eine ausreichende Bearbeitung der Übungsaufgaben erfolgt ist (50%)
  - bei ausreichender Leistung in der Klausur wird der Schein vergeben
  - Bei guter Mitarbeit in den Übungsaufgaben ist die Klausur leicht zu bestehen.

7.2 Studienbegleitende Prüfung

- ★ Keine verpflichtende Abgabe von Übungsaufgaben
  - ◆ aber dringend empfohlen
- ★ Klausur am Semesterende (120 min)
  - ◆ für die Note in der Klausur werden entsprechend Leistungspunkte bzw. Credit Points vergeben
  - ◆ Klausur enthält neben dem Stoff der Übungsaufgaben auch Stoff der Vorlesung, vor allem solchen, der mit den Übungsaufgaben in Zusammenhang steht.
- ▲ Klausurtermin für beide Klausurvarianten
  - ◆ Donnerstag, 7. Februar 2002, 16:00 Uhr

B Einführung

1 Warum Systemprogrammierung I?

- Rasche Einarbeitung in spezielle Systeme
  - MVS, BS2000, VM, Solaris, Unix, Windows NT, Windows 95/98, MS/DOS
- Strukturierung komplexer Programmsysteme
  - Unterteilung in interagierende Komponenten
- Konzeption und Implementierung spezialisierter Systeme
  - Eingebettete Systeme *(Embedded Systems)*
  - Automatisierungssysteme
- Erstellung fehlertoleranter Systeme
- Verständnis für Abläufe im Betriebssystem
  - Ökonomische Nutzung der Hardware
  - Laufzeitoptimierung anspruchsvoller Anwendungen

1.1 Phänomene der Speicherverwaltung

- Beispiel: Initialisierung von großen Matrizen
  - Variante 1:

```
#define DIM 6000

int main()
{
    register long i, j;
    static long matrix[DIM][DIM];

    for( i= 0; i< DIM; i++ )
        for( j= 0; j< DIM; j++ )
            matrix[i][j]= 1;

    exit(0);
}
```

1.1 Phänomene der Speicherverwaltung (2)

- Beispiel: Initialisierung von großen Matrizen
  - Variante 2:

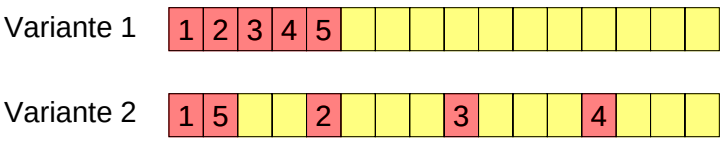
```
#define DIM 6000

int main()
{
    register long i, j;
    static long matrix[DIM][DIM];

    for( j= 0; j< DIM; j++ )
        for( i= 0; i< DIM; i++ )
            matrix[i][j]= 1;

    exit(0);
}
```

  - Schleifen sind vertauscht

1.1 Phänomene der Speicherverwaltung (3)

- Messergebnisse (Sun UltraSparc 1, 128M Hauptspeicher)
  - Variante 1:
    User time= 3,69 sec; System time= 1,43 sec; Gesamtzeit= 22,03 sec
  - Variante 2:
    User time= 21,86 sec; System time= 2,33 sec; Gesamtzeit= 86,39 sec
- Ursachen
  - Variante 1 geht sequentiell durch den Speicher
  - Variante 2 greift versetzt ständig auf den gesamten Speicherbereich zu

Beispiel: **`matrix[4][4]`** und die ersten fünf Zugriffe



1.1 Phänomene der Speicherverwaltung (4)

- Ursachen
  - ◆ Logischer Adressraum
    - Benutzte Adressen sind nicht die physikalischen Adressen
    - Abbildung wird durch Hardware auf Seitenbasis vorgenommen (Seitenadressierung)
    - Variante 2 hat weniger Lokalität, d.h. benötigt häufig wechselnde Abbildungen
  - ◆ Virtueller Speicher
    - Möglicher Adressraum ist größer als physikalischer Speicher
    - Auf Seitenbasis werden Teile des benötigten Speichers ein- und ausgelagert
    - bei Variante 2 muss viel mehr Speicher ein- und ausgelagert werden

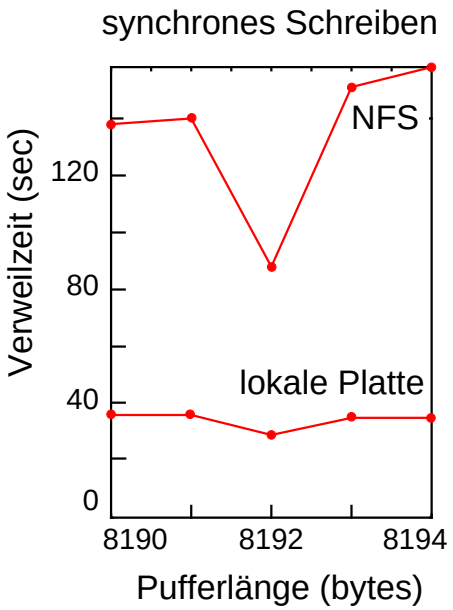
1.2 Phänomene des Dateisystems

- Beispiel: Sequentielles Schreiben mit unterschiedlicher Pufferlänge

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUFLEN 8191

int main()
{
    static char buffer[BUFLEN];
    int i, fd= open( "filename",
                     O_CREAT|O_TRUNC|O_WRONLY|O_SYNC,
                     S_IRUSR|S_IWUSR );

    for( i= 0; i < 1000; i++ )
        write( fd, buffer, BUFLEN );

    exit(0);
}
```

1.2 Phänomene des Dateisystems (2)

- Messergebnisse



1.2 Phänomene des Dateisystems (3)

- Ursachen
  - Synchrones Schreiben erfordert sofortiges Rausschreiben der Daten auf Platte (nötig beispielsweise, wenn hohe Fehlertoleranz gefordert wird – Platte ist immer auf dem neuesten Stand)
  - 8192 ist ein Vielfaches der Blockgröße der Plattenblocks
  - kleine Abweichungen von der Blockgröße erfordern bereits zusätzliche Blocktransfers

2 Überblick über die Vorlesung

★ Inhaltsübersicht

3 Was sind Betriebssysteme?

■ DIN 44300

◆ „...die Programme eines digitalen Rechensystems, die zusammen mit den Eigenschaften der Rechenanlage die **Basis der möglichen Betriebsarten** des digitalen Rechensystems bilden und die insbesondere die **Abwicklung von Programmen steuern und überwachen.**“

■ Tanenbaum

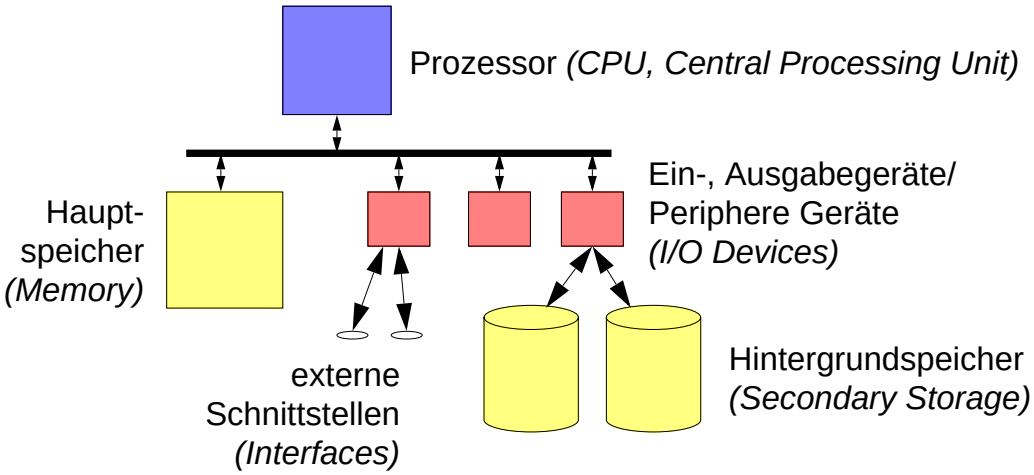
◆ „...eine Software-Schicht ..., die alle Teile des Systems verwaltet und dem Benutzer eine Schnittstelle oder eine *virtuelle Maschine* anbietet, die einfacher zu verstehen und zu programmieren ist [als die nackte Hardware].“

3 Was sind Betriebssysteme? (2)

- Silberschatz/Galvin
  - „... ein Programm, das als Vermittler zwischen Rechnernutzer und Rechner-Hardware fungiert. Der Sinn des Betriebssystems ist eine Umgebung bereitzustellen, in der Benutzer bequem und effizient Programme ausführen können.“
- Brinch Hansen
  - „... der Zweck eines Betriebssystems [liegt] in der Verteilung von Betriebsmitteln auf sich bewerbende Benutzer.“
- ★ Zusammenfassung
  - Software zur Betriebsmittelverwaltung
  - Bereitstellung von Grundkonzepten zur statischen und dynamischen Strukturierung von Programmsystemen

3.1 Verwaltung von Betriebsmitteln

- Betriebsmittel



3.1 Verwaltung von Betriebsmitteln (2)

- Resultierende Aufgaben
  - Multiplexen von Betriebsmitteln für mehrere Benutzer bzw. Anwendungen
  - Schaffung von Schutzumgebungen
- Ermöglichen einer koordinierten gemeinsamen Nutzung von Betriebsmitteln, klassifizierbar in
  - aktive, zeitlich aufteilbare (Prozessor)
  - passive, nur exklusiv nutzbare (periphere Geräte, z.B. Drucker u.Ä.)
  - passive, räumlich aufteilbare (Speicher, Plattenspeicher u.Ä.)
- Unterstützung bei der Fehlererholung

3.2 Schnittstellen

- Betriebssystem soll Benutzervorstellungen auf die Maschinengegebenheiten abbilden
  - Bereitstellung geeigneter Abstraktionen und Schnittstellen für

    Benutzer:
    Dialogbetrieb, graphische Benutzeroberflächen

    Anwendungsprogrammierer:
    Programmiersprachen, Modularisierungshilfen, Interaktionsmodelle (Programmiermodell)

    Systemprogrammierer:
    Werkzeuge zur Wartung und Pflege

    Operateure:
    Werkzeuge zur Gerätebedienung und Anpassung von Systemstrategien

3.2 Schnittstellen (2)

Administratoren:

Werkzeuge zur Benutzerverwaltung, langfristige Systemsteuerung

Programme:

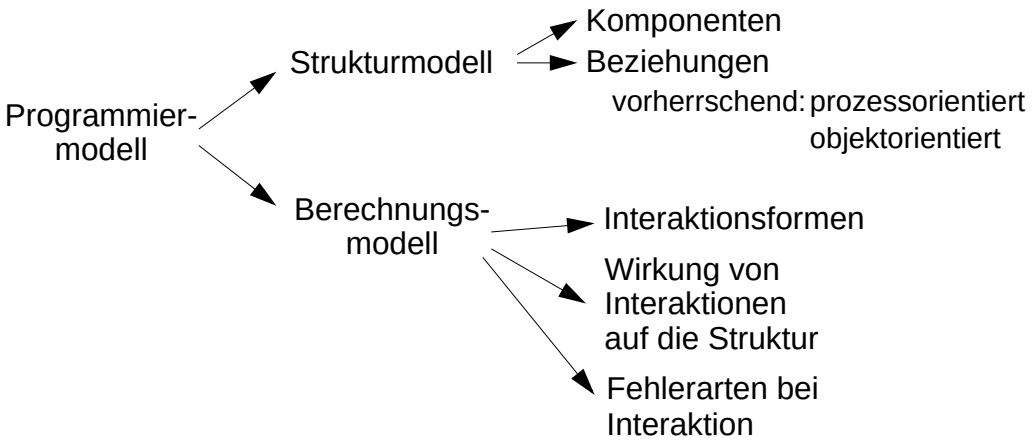
„*Supervisor Calls* (SVC)“,
„*Application Programmer Interface* (API)“

Hardware:

Gerätetreiber

3.3 Programmiermodelle

- Betriebsystem realisiert ein Programmiermodell
  - Keine Notwendigkeit genauer Kenntnisse über Hardwareeigenschaften und spezielle Systemsoftwarekomponenten
  - Schaffung einer begrifflichen Basis zur Strukturierung von Programmsystemen und ihrer Ablaufsteuerung



3.3 Programmiermodelle (2)

- Beispiele für Strukturkomponenten
  - Dateien (Behälter zur langfristigen Speicherung von Daten)
  - Prozesse (in Ausführung befindliche Programme)
  - Klassen (Vorlagen zur Bildung von Instanzen)
  - Instanzen/Objekte
  - Prozeduren
  - Sockets (Kommunikationsendpunkte, „Kommunikationssteckdosen“)
  - Pipes (Nachrichtenkanäle)
- Beispiele für Beziehungen
  - A kann B referenzieren, beauftragen, aufrufen, modifizieren
  - Pipe P verbindet A und B

3.3 Programmiermodelle (3)

- Beispiele für Interaktionsformen
  - Prozedur-(Methoden-)Aufruf
  - Nachrichtenaustausch
  - Gemeinsame Speichernutzung
- Wirkung von Interaktionen auf die Struktur
  - Erzeugung und Tilgung von Prozessen
  - Instanziierung von Objekten
- Fehlerarten bei Interaktion
  - Verlust, Wiederholung oder Verspätung von Nachrichten
  - Abbruch aufgerufener Methoden, Ausnahmebehandlung

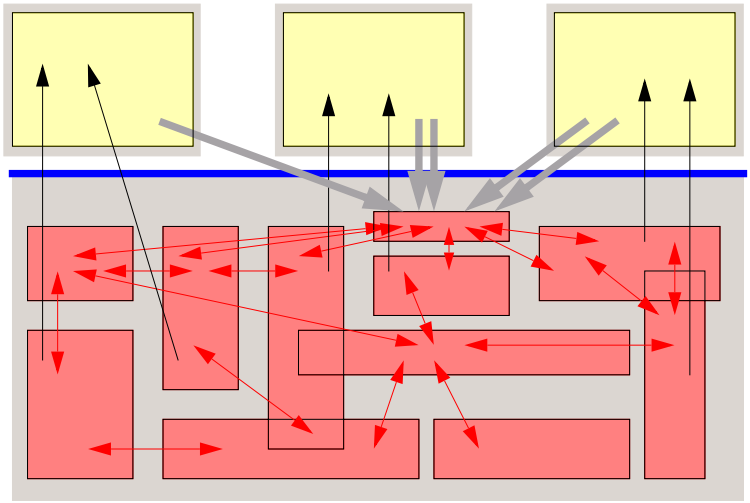
3.4 Ablaufmodelle

- Betriebssystem realisiert eine Ablaufumgebung
- Bereitstellung von Hilfsmitteln zur Bearbeitung von Benutzerprogrammen und zur Steuerung ihrer Abläufe.
  - Laden und Starten von Programmen
  - Überwachung des Programmablaufs
  - Beenden und Eliminieren von Programmen
  - Abrechnung *(Accounting)*

4 Betriebssystemarchitekturen

- Umfang zehntausende bis mehrere Millionen Befehlszeilen
  - Strukturierung hilfreich
- Verschiedene Strukturkonzepte
  - monolithische Systeme
  - geschichtete Systeme
  - Minimalkerne
  - offene objektorientierte Systeme

4.1 Monolithische Systeme



billige, ungeschützte Interaktion

teuere, geschützte Interaktion

Adressraum

Anwendung

Betriebssystemfunktion

4.1 Monolithische Systeme (2)

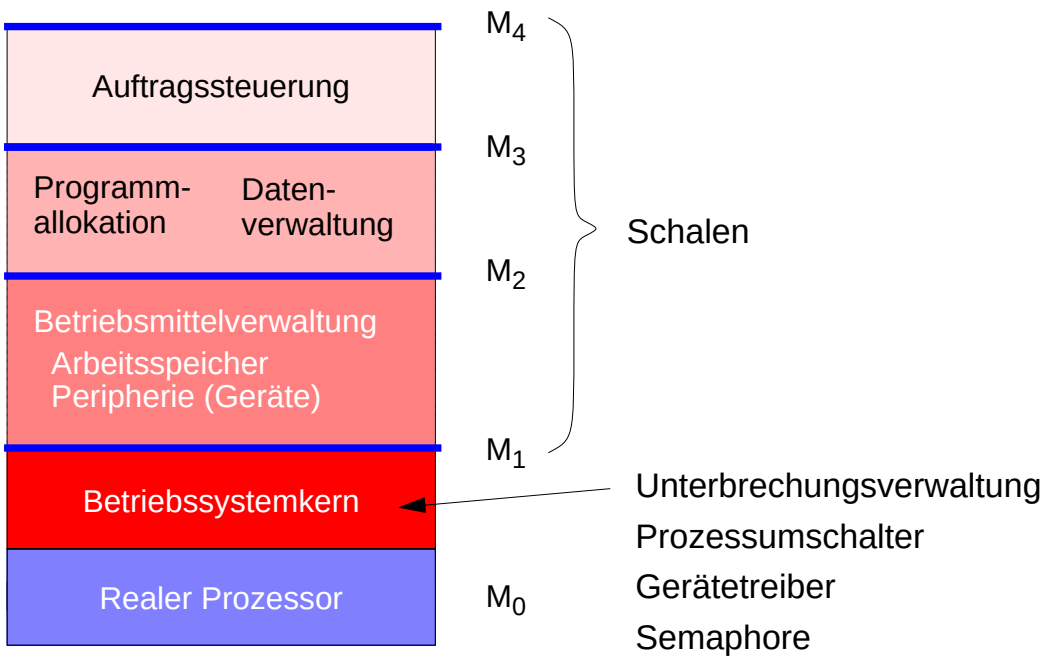
★ Vorteile

- ◆ Effiziente Kommunikation und effizienter Datenzugriff innerhalb des Kerns
- ◆ Privilegierte Befehle jederzeit ausführbar

▲ Nachteile

- ◆ Keine interne Strukturierung (änderungsunfreundlich, fehleranfällig)
- ◆ Kein Schutz zwischen Kernkomponenten (Problem: zugekaufte Treiber)

4.2 Geschichtete Systeme

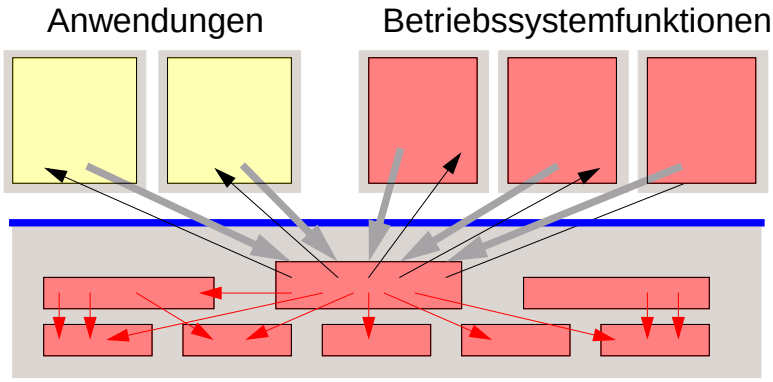


4.2 Geschichtete Systeme (2)

★ Vorteile
- ◆ Schutz zwischen verschiedenen BS-Teilen
- ◆ Interne Strukturierung

▲ Nachteile
- ◆ Mehrfacher Schutzraumwechsel ist teuer
- ◆ Unflexibler und nur einseitiger Schutz (von unten nach oben)

4.3 Minimalkerne



billige, ungeschützte Interaktion

teuere, geschützte Interaktion

Adressraum

Anwendung

Betriebssystemfunktion

4.3 Minimalkerne (2)

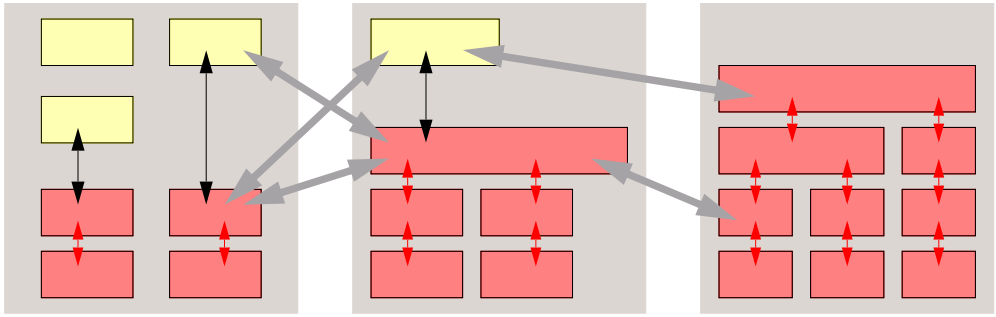
★ Vorteile

- ◆ Gute Modularisierung
- ◆ Schutz der Komponenten voreinander

▲ Nachteil

- ◆ Kommunikation zwischen Modulen ist teuer

4.4 Objektbasierte, offene Systeme



billige, durch Objektkapselung geschützte Interaktion

teuere, durch Adressraumgrenze geschützte Interaktion

Adressraum

Anwendungsobjekte

Betriebssystemobjekte

- Sicherung der Modulgrenzen durch Programmiermodell und Software
  - z.B. Objektorientierung und Byte-Code-Verifier in Java

4.4 Objektbasierte, offene Systeme (2)

- ★ Vorteile
  - Schutz auf mehreren Ebenen (Sprache, Code-Prüfung, Adressraum)
  - Modularisierung und Effizienz möglich
- ▲ Nachteile
  - Komplexes Sicherheitsmodell

5 Geschichtliche Entwicklung

5.1 1950–1960

1950

- Einströmige Stapelsysteme
  (Single-stream batch processing systems)
  Aufträge zusammen mit allen Daten werden übergeben und sequentiell bearbeitet

- Steuerung durch Auftragsabwickler
  (Resident monitor, Job monitor)
  Hilfsmittel: Assembler, Compiler, Binder und Lader, Programmbibliotheken

1960

5.2 1960–1965

1960

- Autonome periphere Geräte → Überlappung von Programmbearbeitung und Datentransport zw. Arbeitsspeicher und peripheren Geräten möglich
  - Wechselpufferbetrieb (abwechselndes Nutzen zweier Puffer)
  - Mehrprogrammbetrieb *(Multiprogramming)*
  - Spooling *(Simultaneous peripheral operation on-line)*

- Mehrere Programme müssen gleichzeitig im Speicher sein → Auslagern von Programmen auf Sekundärspeicher
- Programme müssen während des Ablaufs verlagerbar sein *(Relocation problem)*

- Echtzeitdatenverarbeitung *(Real-time processing)*, d.h. enge Bindung von Ein- und Ausgaben an die physikalische Zeit

1965

5.3 1965–1970

1965

OS/360

- Umsetzung von Programmadressen in Speicherorte zur Laufzeit: Segmentierung, Seitenadressierung *(Paging)*
- Virtueller Adressraum: Seitentausch *(Paging)*
  Seiten werden je nach Zugriff ein- und ausgelagert
- Interaktiver Betrieb *(Interactive processing, Dialog mode)*

THE

- Mehrbenutzerbetrieb, Teilnehmersysteme *(Time sharing)*

MULTICS

- Problem: Kapselung von Prozessen und Dateien → geschützte Adressraum, Zugriffsschutz auf Dateien
- Dijkstra: Programmsysteme als Menge kooperierender Prozesse (heute *Client-Server*)

UNIX

- Problem: Prozessinteraktion bei gekapselten Prozessen → Nachrichtensysteme zur Kommunikation, gemeinsamer Speicher zur Kooperation

1970

5.4 1970–1975

1970

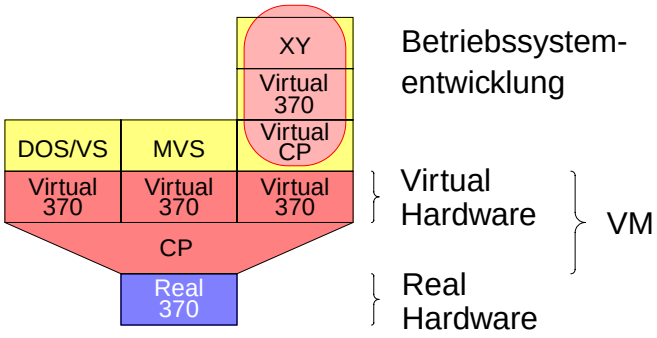
VM

- Modularisierung:
  Datenkapselung, Manipulation durch Funktionen (nach Parnas)

Hydra

- Virtuelle Maschinen: Koexistenz verschiedener Betriebssysteme im gleichen Rechner

MVS



- Symmetrische Multiprozessoren: HYDRA
  - Zugangskontrolle zu Instanzen durch Capabilities
  - Trennung von Strategie und Mechanismus

1975

5.5 1975–1985

1975

- Vernetzung, Protokolle (z.B. TCP/IP)
- Verteilte Systeme
- Newcastle Connection
- Fernaufruf *(Remote procedure call, RPC)*

LOCUS

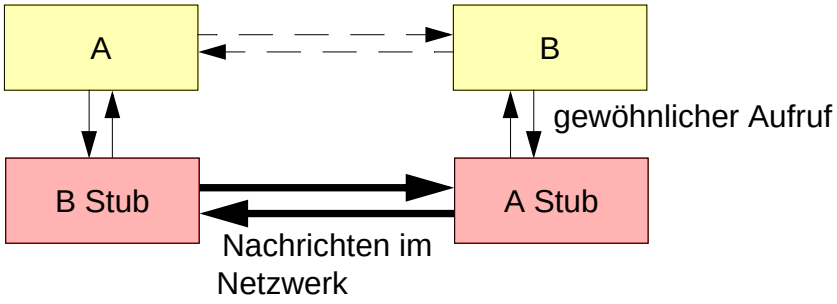
1980

MS-DOS

NC

EDEN

1985



5.6 1985–1999

1985

OS/2

- Kryptographie
- Authentifizierung und Authentisierung
- Objektorientierte Systeme

Mach 3.0

- Parallele Systeme
- Mikrokerne

1990

Windows

Spring

Win NT

- Objektorientierte Mikrokerne
- Internet, Multimedia

1995

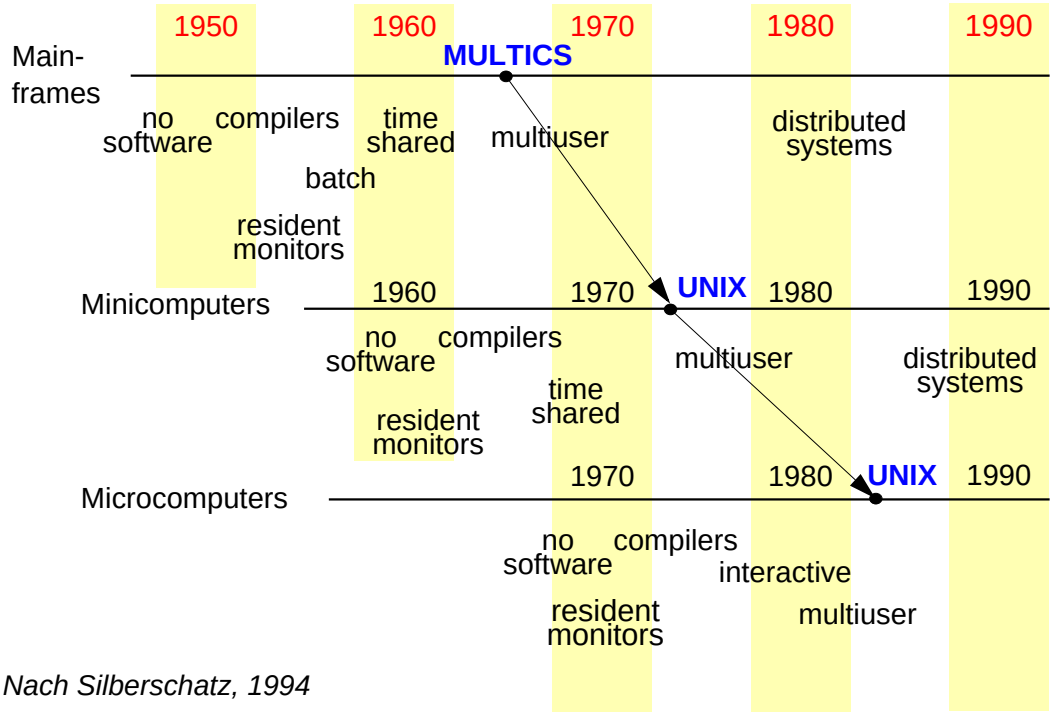
5.7 1995–1999

1995
SPIN
Exokernel
L4
Linux

1999

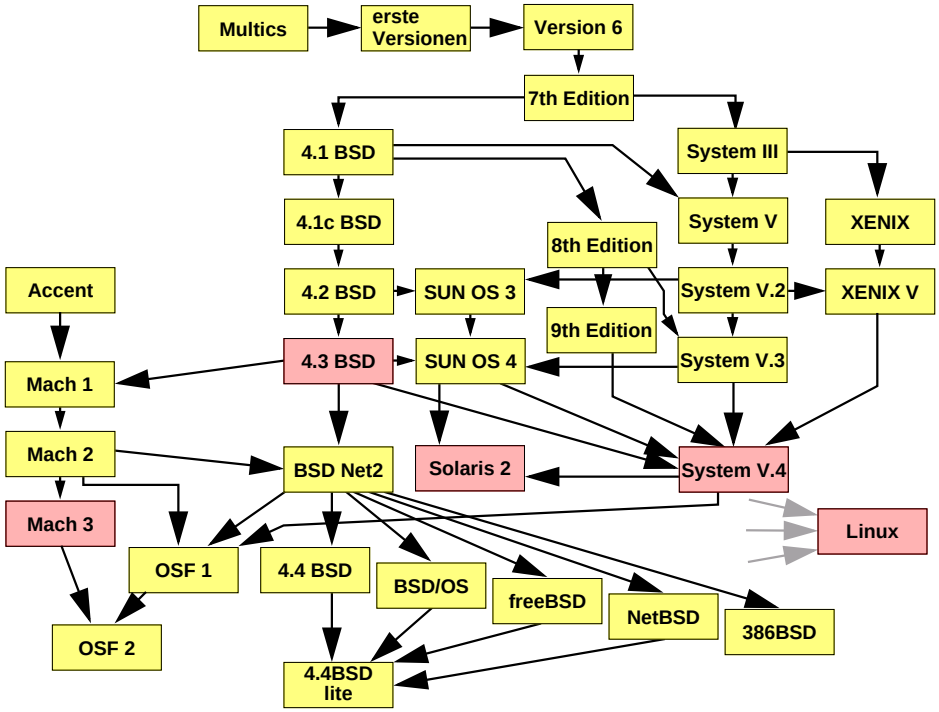
- Echtzeitscheduling in Standardbetriebssystemen
- 64bit-Adressierung

5.8 Migration von Konzepten



Nach Silberschatz, 1994

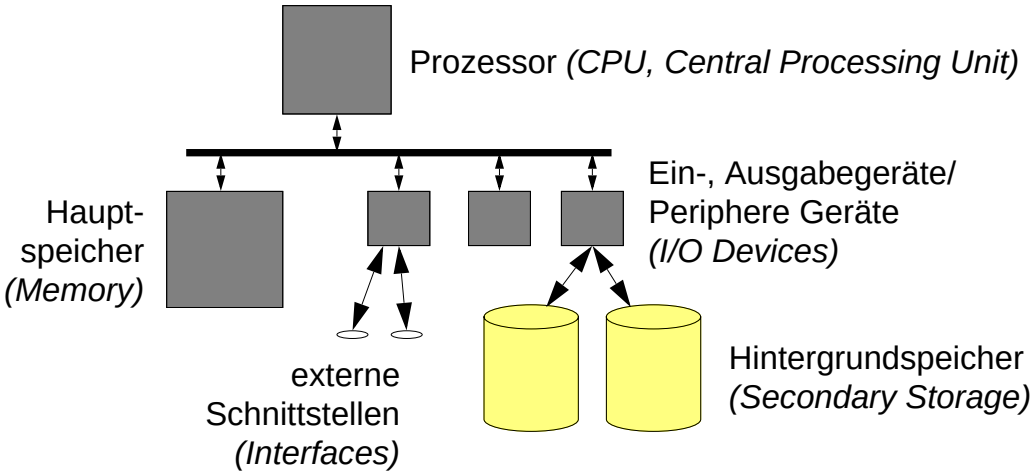
5.9 UNIX Entwicklung



C Dateisysteme

C Dateisysteme

- Einordnung

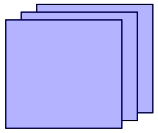


C Dateisysteme (2)

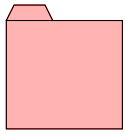
- Dateisysteme speichern Daten und Programme persistent in Dateien
  - ◆ Betriebssystemabstraktion zur Nutzung von Hintergrundspeichern (z.B. Platten, CD-ROM, Floppy Disk, Bandlaufwerke)
    - Benutzer muss sich nicht um die Ansteuerungen verschiedener Speichermedien kümmern
    - einheitliche Sicht auf den Sekundärspeicher
- Dateisysteme bestehen aus
  - ◆ Dateien *(Files)*
  - ◆ Verzeichnissen, Katalogen *(Directories)*
  - ◆ Partitionen *(Partitions)*

C Dateisysteme (3)

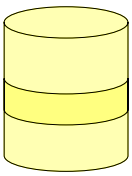
- Datei
  - ◆ speichert Daten oder Programme



- Verzeichnis
  - ◆ fasst Dateien (u. Verzeichnisse) zusammen
  - ◆ erlaubt Benennung der Dateien
  - ◆ enthält Zusatzinformationen zu Dateien



- Partitionen
  - ◆ eine Menge von Verzeichnissen und deren Dateien
  - ◆ Sie dienen zum physischen oder logischen Trennen von Dateimengen.
    - *physisch:* Festplatte, Diskette
    - *logisch:* Teilbereich auf Platte oder CD



1 Dateien

- Kleinste Einheit, in der etwas auf den Hintergrundspeicher geschrieben werden kann.

1.1 Dateiattribute

- *Name* — Symbolischer Name, vom Benutzer les- und interpretierbar
  - ◆ z.B. **`AUTOEXEC.BAT`**
- *Typ* — Für Dateisysteme, die verschiedene Dateitypen unterscheiden
  - ◆ z.B. sequenzielle Datei, zeichenorientierte Datei, satzorientierte Datei
- *Ortsinformation* — Wo werden die Daten physisch gespeichert?
  - ◆ Gerätenummer, Nummern der Plattenblocks

1.1 Dateiattribute (2)

- *Größe* — Länge der Datei in Größeneinheiten (z.B. Bytes, Blöcke, Sätze)
  - ◆ steht in engem Zusammenhang mit der Ortsinformation
  - ◆ wird zum Prüfen der Dateigrenzen z.B. beim Lesen benötigt
- *Zeitstempel* — z.B. Zeit und Datum der Erstellung, letzten Änderung
  - ◆ unterstützt Backup, Entwicklungswerkzeuge, Benutzerüberwachung etc.
- *Rechte* — Zugriffsrechte, z.B. Lese-, Schreibberechtigung
  - ◆ z.B. nur für den Eigentümer schreibbar, für alle anderen nur lesbar
- *Eigentümer* — Identifikation des Eigentümers
  - ◆ eventuell eng mit den Rechten verknüpft
  - ◆ Zuordnung beim Accounting (Abrechnung des Plattenplatzes)

1.2 Operationen auf Dateien

- Erzeugen *(Create)*
  - ◆ Nötiger Speicherplatz wird angefordert.
  - ◆ Verzeichniseintrag wird erstellt.
  - ◆ Initiale Attribute werden gespeichert.
- Schreiben *(Write)*
  - ◆ Identifikation der Datei
  - ◆ Daten werden auf Platte transferiert.
  - ◆ eventuelle Anpassung der Attribute, z.B. Länge
- Lesen *(Read)*
  - ◆ Identifikation der Datei
  - ◆ Daten werden von Platte gelesen.

1.2 Operationen auf Dateien (2)

- Positionieren des Schreib-/Lesezeigers *(Seek)*
  - Identifikation der Datei
  - In vielen Systemen wird dieser Zeiger implizit bei Schreib- und Leseoperationen positioniert.
  - Ermöglicht explizites Positionieren
- Verkürzen *(Truncate)*
  - Identifikation der Datei
  - Ab einer bestimmten Position wird der Inhalt entfernt (evtl. kann nur der Gesamtinhalt gelöscht werden).
  - Anpassung der betroffenen Attribute
- Löschen *(Delete)*
  - Identifikation der Datei
  - Entfernen der Datei aus dem Katalog und Freigabe der Plattenblocks

2 Verzeichnisse / Kataloge

- Ein Verzeichnis gruppiert Dateien und evtl. andere Verzeichnisse
- Gruppierungsalternativen:
  - Verknüpfung mit der Benennung
    - Verzeichnis enthält Namen und Verweise auf Dateien und andere Verzeichnisse, z.B. *UNIX, MS-DOS*
  - Gruppierung über Bedingung
    - Verzeichnis enthält Namen und Verweise auf Dateien, die einer bestimmten Bedingung gehorchen
      z.B. gleiche Gruppennummer in *CP/M*
      z.B. eigenschaftsorientierte und dynamische Gruppierung in *BeOS-BFS*
- Verzeichnis ermöglicht das Auffinden von Dateien
  - Vermittlung zwischen externer und interner Bezeichnung (Dateiname — Plattenblöcken)

2.1 Operationen auf Verzeichnissen

- Auslesen der Einträge *(Read, Read Directory)*
  - Daten des Verzeichnisinhalts werden gelesen und meist eintragsweise zurückgegeben
- Erzeugen und Löschen der Einträge erfolgt implizit mit der zugehörigen Dateioperation
- Erzeugen und Löschen von Verzeichnissen *(Create and Delete Directory)*

2.2 Attribute von Verzeichnissen

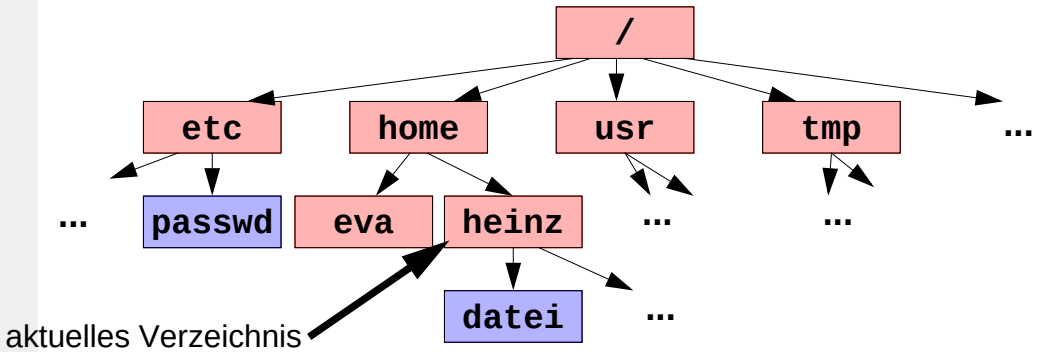
- Die meisten Dateiattribute treffen auch für Verzeichnisse zu
  - Name, Ortsinformationen, Größe, Zeitstempel, Rechte, Eigentümer

3 Beispiel: UNIX (Sun-UFS)

- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
  - Zugriffsrechte: lesbar, schreibbar, ausführbar
- Verzeichnis
  - baumförmig strukturiert
    - Knoten des Baums sind Verzeichnisse
    - Blätter des Baums sind Verweise auf Dateien *(Links)*
  - jedem UNIX-Prozess ist zu jeder Zeit ein aktuelles Verzeichnis *(Current Working Directory)* zugeordnet
  - Zugriffsrechte: lesbar, schreibbar, durchsuchbar, „nur“ erweiterbar

3.1 Pfadnamen

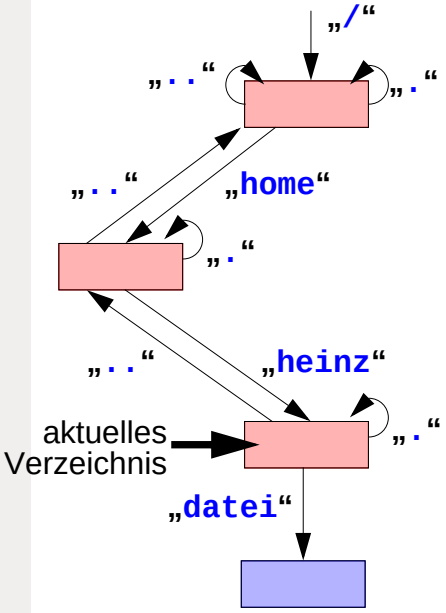
- Baumstruktur



- Pfade
  - z.B. „**/home/heinz/datei**“, „**/tmp**“, „**datei**“
  - „**/**“ ist Trennsymbol *(Slash)*; beginnender „**/**“ bezeichnet Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellem Verzeichnis

3.1 Pfadnamen (2)

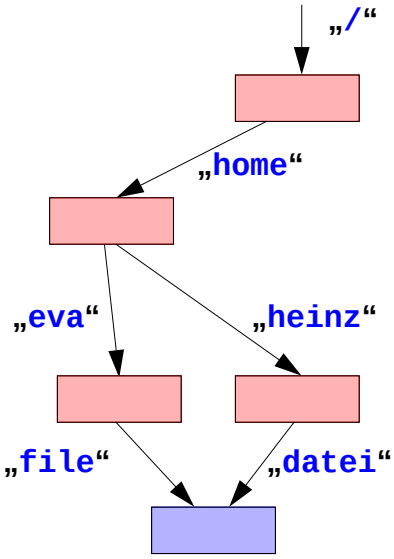
- Eigentliche Baumstruktur



- benannt sind nicht Dateien und Verzeichnisse, sondern die Verbindungen zwischen ihnen
  - Verzeichnisse und Dateien können auf verschiedenen Pfaden erreichbar sein z.B. **../heinz/datei** und **/home/heinz/datei**
  - Jedes Verzeichnis enthält einen Verweis auf sich selbst („**.**“) und einen Verweis auf das darüberliegende Verzeichnis im Baum („**..**“)

3.1 Pfadnamen (3)

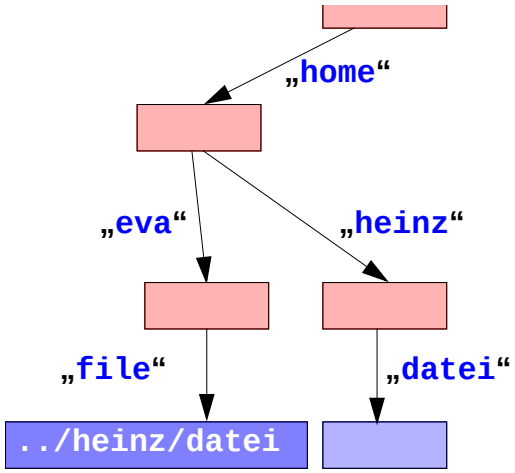
- Links *(Hard Links)*
  - Dateien können mehrere auf sich zeigende Verweise besitzen, sogenannte Hard-Links (nicht jedoch Verzeichnisse)



  - Die Datei hat zwei Einträge in verschiedenen Verzeichnissen, die völlig gleichwertig sind:
    `/home/eva/file`
    `/home/heinz/datei`
  - Datei wird erst gelöscht, wenn letzter Link gekappt wird.

3.1 Pfadnamen (4)

- Symbolische Namen *(Symbolic Links)*
  - Verweise auf einen anderen Pfadnamen (sowohl auf Dateien als auch Verzeichnisse)
  - Symbolischer Name bleibt auch bestehen, wenn Datei oder Verzeichnis nicht mehr existiert



  - Symbolischer Name enthält einen neuen Pfadnamen, der vom FS interpretiert wird.

3.2 Eigentümer und Rechte

- Eigentümer
  - Jeder Benutzer wird durch eindeutige Nummer (UID) repräsentiert
  - Ein Benutzer kann einer oder mehreren Benutzergruppen angehören, die durch eine eindeutige Nummer (GID) repräsentiert werden
  - Eine Datei oder ein Verzeichnis ist genau einem Benutzer und einer Gruppe zugeordnet
- Rechte auf Dateien
  - Lesen, Schreiben, Ausführen (nur vom Eigentümer veränderbar)
  - einzeln für den Eigentümer, für Angehörige der Gruppe und für alle anderen einstellbar
- Rechte auf Verzeichnissen
  - Lesen, Schreiben (Löschen u. Anlegen von Dateien etc.), Durchgangsrecht
  - Schreibrecht ist einschränkbar auf eigene Dateien („nur erweiterbarer“)

3.3 Dateien

- Basisoperationen
  - Öffnen einer Datei

```
int open( const char *path, int oflag, [mode_t mode] );
```

    - Rückgabewert ist ein Filedescriptor, mit dem alle weiteren Dateioperationen durchgeführt werden müssen.
    - Filedescriptor ist nur prozesslokal gültig.

  - Sequentielles Lesen und Schreiben

```
ssize_t read( int fd, void *buf, size_t nbytes );
```

    Gibt die Anzahl gelesener Zeichen zurück

```
ssize_t write( int fd, void *buf, size_t nbytes );
```

    Gibt die Anzahl geschriebener Zeichen zurück

3.3 Dateien (2)

- Basisoperationen (2)
  - Schließen der Datei

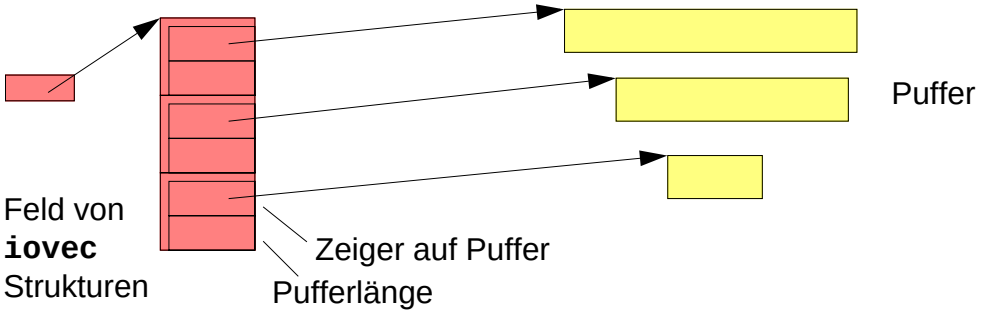
```
int close( int fd );
```

- Fehlermeldungen
  - Anzeige durch Rückgabe von **`-1`**
  - Variable **`int errno`** enthält Fehlercode
  - Funktion **`perror("")`** druckt Fehlermeldung bzgl. **`errno`** auf die Standard-Ausgabe

3.3 Dateien (2)

- Weitere Operationen
  - Lesen und Schreiben in Pufferlisten

```
int readv( int fd, const struct iovec *iov, int iovcnt );
int writev( int fd, const struct iovec *iov, int iovcnt );
```



  - Positionieren des Schreib-, Lesezeigers

```
off_t lseek( int fd, off_t offset, int whence );
```

3.3 Dateien (3)

- Attribute einstellen
  - Länge

```
int truncate( const char *path, off_t length );
int ftruncate( int fd, off_t length );
```

  - Zugriffs- und Modifikationszeiten

```
int utimes( const char *path, const struct timeval *tvp );
```

  - Implizite Maskierung von Rechten

```
mode_t umask( mode_t mask );
```

  - Eigentümer und Gruppenzugehörigkeit

```
int chown( const char *path, uid_t owner, gid_t group );
int lchown( const char *path, uid_t owner, gid_t group );
int fchown( int fd, uid_t owner, gid_t group );
```


3.3 Dateien (4)

- Zugriffsrechte

```
int chmod( const char *path, mode_t mode );
int fchmod( int fd, mode_t mode );
```

- Alle Attribute abfragen

```
int stat( const char *path, struct stat *buf );
```

Alle Attribute von **path** ermitteln (folgt symbolischen Links)

```
int lstat( const char *path, sturct stat *buf );
```

Wie **stat**, folgt aber symbolischen Links nicht

```
int fstat( int fd, struct stat *buf );
```

Wie **stat**, aber auf offene Datei

3.4 Verzeichnis

- Verzeichnisse verwalten
  - Erzeugen

```
int mkdir( const char *path, mode_t mode );
```

  - Löschen

```
int rmdir( const char *path );
```

  - Hard Link erzeugen

```
int link( const char *existing, const char *new );
```

  - Symbolischen Namen erzeugen

```
int symlink( const char *path, const char *new );
```

  - Verweis/Datei löschen

```
int unlink( const char *path );
```


3.4 Verzeichnisse (2)

- Verzeichnisse auslesen
  - Öffnen, Lesen und Schließen wie eine normale Datei
  - Interpretation der gelesenen Zeichen ist jedoch systemabhängig, daher wurde eine systemunabhängige Schnittstelle zum Lesen definiert:

```
int getdents( int fildes, struct dirent *buf,
              size_t nbyte );
```

  - Zum einfacheren Umgang mit Katalogen gibt es Bibliotheksfunktionen:

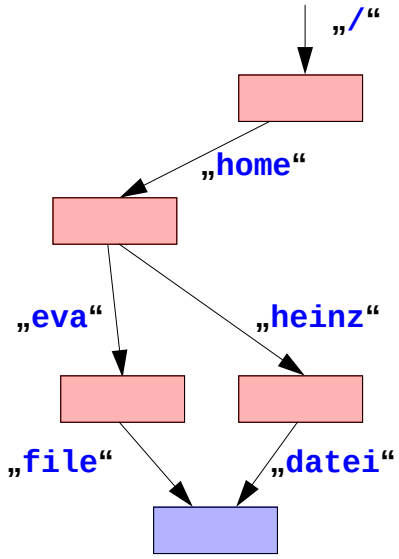
```
DIR *opendir( const char *path );
struct dirent *readdir( DIR *dirp );
int closedir( DIR *dirp );
long telldir( DIR *dirp );
void seekdir( DIR *dirp, long loc );
```

- Symbolische Namen auslesen

```
int readlink( const char *path, void *buf, size_t bufsiz );
```

3.5 Inodes

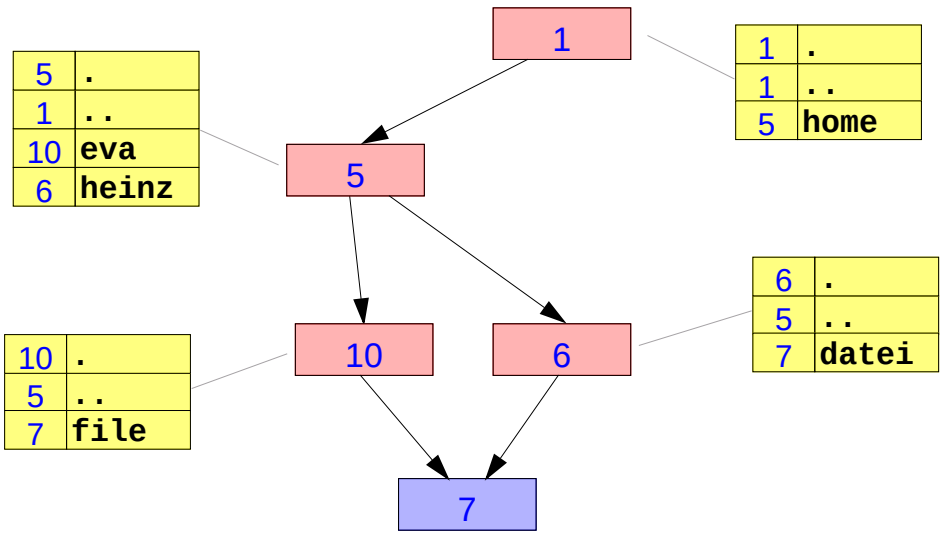
- Attribute einer Datei und Ortsinformationen über ihren Inhalt werden in sogenannten Inodes gehalten
  - Inodes werden pro Partition nummeriert *(Inode Number)*



logischer Dateibaum

3.5 Inodes (2)

- Verzeichnisse enthalten lediglich Paare von Namen und Inode-Nummern



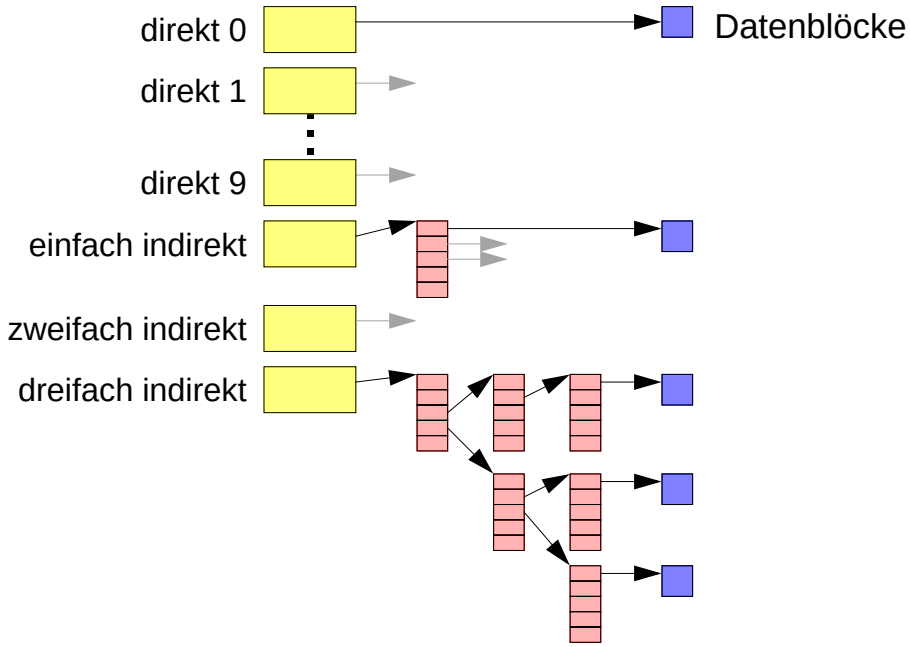
tatsächlich gespeicherter Baum

3.5 Inodes (3)

- Inhalt eines Inodes
  - Inodenummer
  - Dateityp: Verzeichnis, normale Datei, Spezialdatei (z.B. Gerät)
  - Eigentümer und Gruppe
  - Zugriffsrechte
  - Zugriffszeiten: letzte Änderung *(mtime)*, letzer Zugriff *(atime)*, letzte Änderung des Inodes *(ctime)*
  - Anzahl der Hard links auf den Inode
  - Dateigröße (in Bytes)
  - Adressen der Datenblöcke des Datei- oder Verzeichnisinhalts (zehn direkt Adressen und drei indirekte)

3.5 Inodes (4)

- Adressierung der Datenblöcke



3.6 Spezialdateien

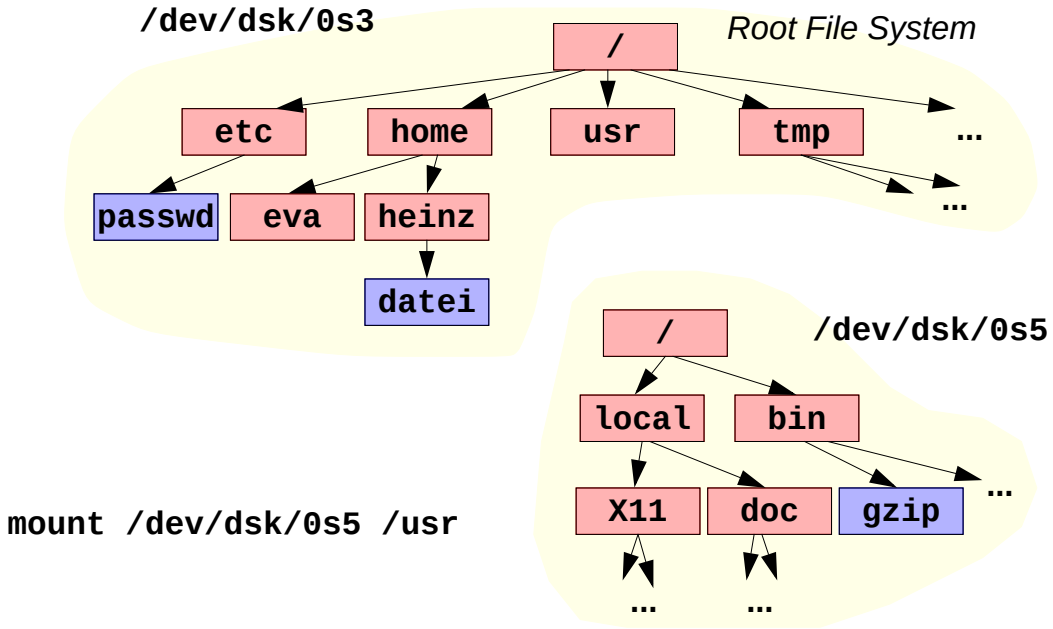
- Periphere Geräte werden als Spezialdateien repräsentiert
  - Geräte können wie Dateien mit Lese- und Schreiboperationen angesprochen werden
  - Öffnen der Spezialdateien schafft eine (evtl. exklusive) Verbindung zum Gerät, die durch einen Treiber hergestellt wird
- Blockorientierte Spezialdateien
  - Plattenlaufwerke, Bandlaufwerke, Floppy Disks, CD-ROMs
- Zeichenorientierte Spezialdateien
  - Serielle Schnittstellen, Drucker, Audiokanäle etc.
  - blockorientierte Geräte haben meist auch eine zusätzliche zeichenorientierte Repräsentation

3.7 Montieren des Dateibaums

- Der UNIX-Dateibaum kann aus mehreren Partitionen zusammenmontiert werden
  - Partition wird Dateisystem genannt *(File system)*
  - wird durch blockorientierte Spezialdatei repräsentiert (z.B. **`/dev/dsk/0s3`**)
  - Das Montieren wird *Mounten* genannt
  - Ausgezeichnetes Dateisystem ist das *Root File System*, dessen Wurzelverzeichnis gleichzeitig Wurzelverzeichnis des Gesamtsystems ist
  - Andere Dateisysteme können mit dem Befehl **`mount`** in das bestehende System hineinmontiert werden

3.7 Montieren des Dateibaums (2)

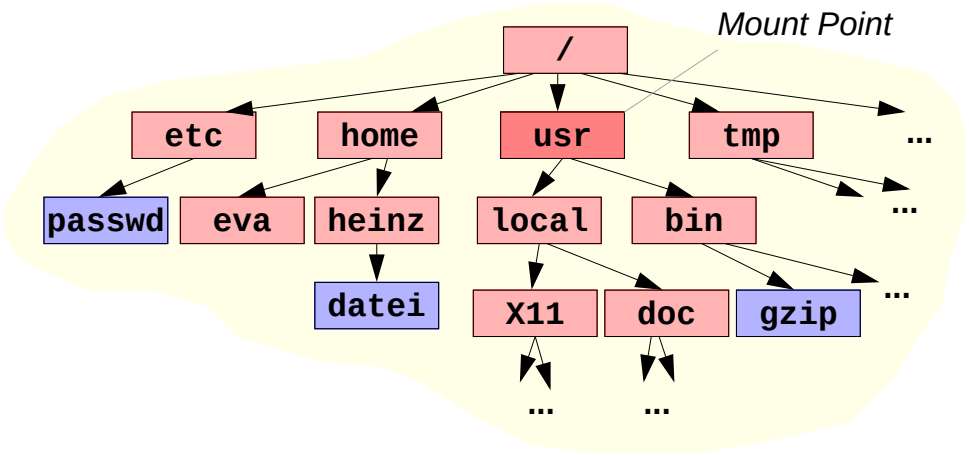
- Beispiel




```
mount /dev/dsk/0s5 /usr
```


3.7 Montieren des Dateibaums (3)

- Beispiel nach Ausführung des Montierbefehls



4 Beispiel: Windows 95 (VFAT, FAT32)

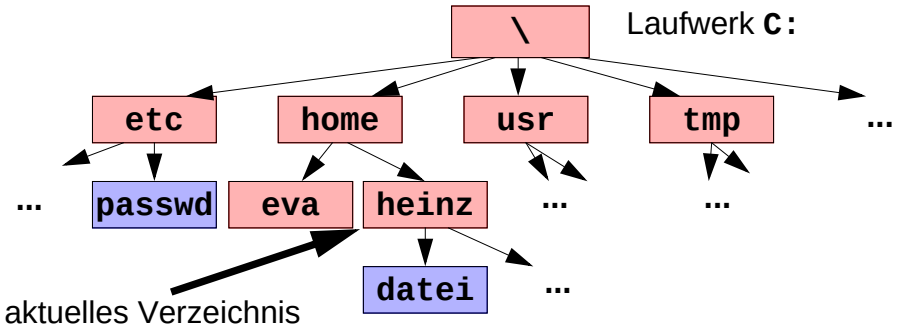
- VFAT = Virtual (!) File Allocation Table (oder FAT32)
  - VFAT: MS-DOS–kompatibles Dateisystem mit Erweiterungen
- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbar
  - Zugriffsrechte: „nur lesbar“, „schreib- und lesebar“

4 Beispiel: Windows 95 (VFAT, FAT32) (2)

- Partitionen heißen Laufwerke
  - Sie werden durch einen Buchstaben dargestellt (z.B. `C:`)
- Verzeichnis
  - baumförmig strukturiert
    - Knoten des Baums sind Verzeichnisse
    - Blätter des Baums sind Dateien
  - jedem Windows-Programm ist zu jeder Zeit ein aktuelles Laufwerk und ein aktuelles Verzeichnis pro Laufwerk zugeordnet
  - Zugriffsrechte: „nur lesbar“, „schreib- und lesbar“

4.1 Pfadnamen

- Baumstruktur



- Pfade
  - z.B. „**C:\home\heinz\datei**“, „**\tmp**“, „**C:datei**“
  - „\“ ist Trennsymbol *(Backslash)*; beginnender „\“ bezeichnet Wurzelverzeichnis; sonst Beginn implizit mit dem aktuellen Verzeichnis
  - beginnt der Pfad ohne Laufwerksbuchstabe wird das aktuelle Laufwerk verwendet

4.1 Pfadnamen (2)

- Namenskonvention
  - Kompatibilitätsmodus: 8 Zeichen Name, 3 Zeichen Erweiterung (z.B. **AUTOEXEC.BAT**)
  - Sonst: 255 Zeichen inklusive Sonderzeichen (z.B. „**Eigene Programme**“)
- Verzeichnisse
  - Jedes Verzeichnis enthält einen Verweis auf sich selbst („**.**“) und einen Verweis auf das darüberliegende Verzeichnis im Baum („**..**“) (Ausnahme Wurzelverzeichnis)
  - keine Hard-Links oder symbolischen Namen

4.2 Rechte

- Rechte pro Datei und Verzeichnis
  - schreib- und lesbar — nur lesbar *(read only)*
- Keine Benutzeridentifikation
  - Rechte garantieren keinen Schutz, da von jedermann veränderbar

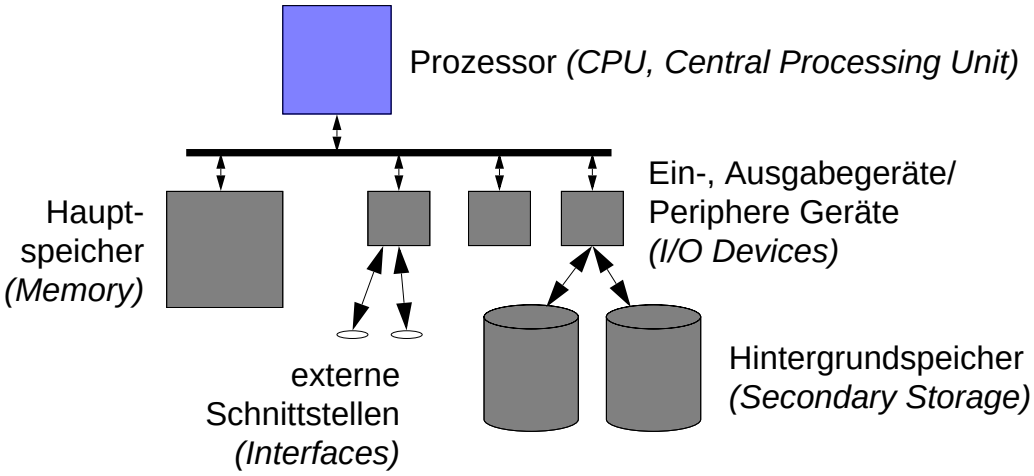
4.3 Dateien

- Attribute
  - Name, Dateilänge
  - Attribute: versteckt *(Hidden)*, archiviert *(Archive)*, Systemdatei *(System)*
  - Rechte
  - Ortsinformation: Nummer des ersten Plattenblocks
  - Zeitstempel: Erzeugung, letzter Schreib- und Lesezugriff

D Prozesse und Nebenläufigkeit

D Prozesse und Nebenläufigkeit

- Einordnung



1 Prozessor

- Register
  - ◆ Prozessor besitzt Steuer- und Vielzweckregister
  - ◆ Steuerregister:
    - Programmzähler *(Instruction Pointer)*
    - Stapelregister *(Stack Pointer)*
    - Statusregister
    - etc.
- Programmzähler enthält Speicherstelle der nächsten Instruktion
  - ◆ Instruktion wird geladen und
  - ◆ ausgeführt
  - ◆ Programmzähler wird inkrementiert
  - ◆ dieser Vorgang wird ständig wiederholt

1 Prozessor (2)

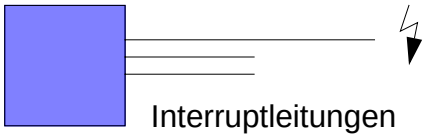
- Beispiel für Instruktionen

```
...
0010 5510000000   movl DS:$10, %ebx
0015 5614000000   movl DS:$14, %eax
001a 8a           addl %eax, %ebx
001b 5a18000000   movl %ebx, DS:$18
...
```

- Prozessor arbeitet in einem bestimmten Modus
  - ◆ Benutzermodus: eingeschränkter Befehlssatz
  - ◆ privilegierter Modus: erlaubt Ausführung privilegierter Befehle
    - Konfigurationsänderungen des Prozessors
    - Moduswechsel
    - spezielle Ein-, Ausgabebefehle

1 Prozessor (3)

- Unterbrechungen *(Interrupts)*



Interruptleitungen

Signalisieren der Unterbrechung *(Interrupt Request; IRQ)*

- ◆ Prozessor unterbricht laufende Bearbeitung und führt eine definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
- ◆ vorher werden alle Register einschließlich Programmzähler gesichert (z.B. auf dem Stack)
- ◆ nach einer Unterbrechung kann der ursprüngliche Zustand wiederhergestellt werden
- ◆ Unterbrechungen werden im privilegierten Modus bearbeitet

1 Prozessor (4)

- Systemaufrufe *(Traps; User Interrupts)*
  - Wie kommt man kontrolliert vom Benutzermodus in den privilegierten Modus?
  - spezielle Befehle zum Eintritt in den privilegierten Modus
  - Prozessor schaltet in privilegierten Modus und führt definierte Befehlsfolge aus (vom privilegierten Modus aus konfigurierbar)
  - solche Befehle werden dazu genutzt die Betriebssystemschnittstelle zu implementieren *(Supervisor Calls)*
  - Parameter werden nach einer Konvention übergeben (z.B. auf dem Stack)

2 Prozesse

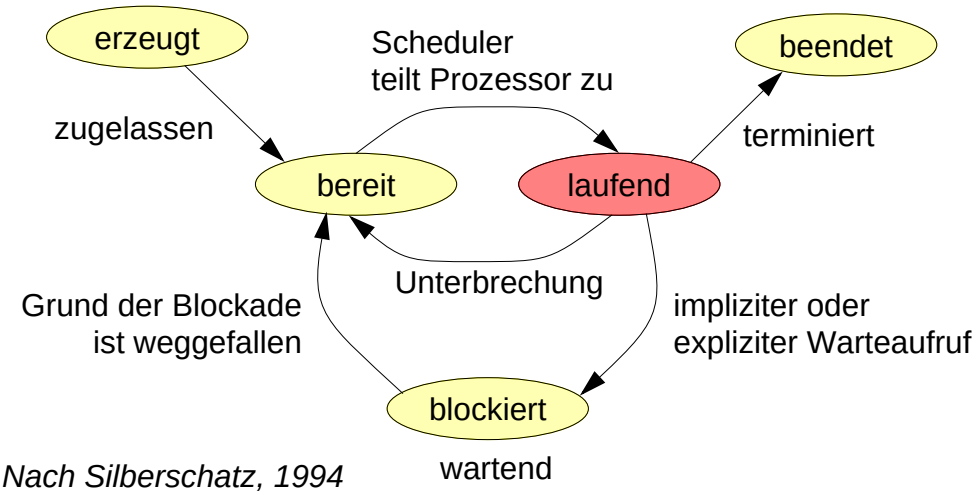
- Stapelsysteme *(Batch Systems)*
  - ein Programm läuft auf dem Prozessor von Anfang bis Ende
- Heutige Systeme *(Time Sharing Systems)*
  - mehrere Programme laufen gleichzeitig
  - Prozessorzeit muss den Programmen zugeteilt werden
  - Programme laufen nebenläufig
- Terminologie
  - **Programm:** Folge von Anweisungen (hinterlegt beispielsweise als Datei auf dem Hintergrundspeicher)
  - **Prozess:** Programm, das sich in Ausführung befindet, und seine Daten (*Beachte*: ein Programm kann sich mehrfach in Ausführung befinden)

2.1 Prozesszustände

- Ein Prozess befindet sich in einem der folgenden Zustände:
  - **Erzeugt** *(New)*
    Prozess wurde erzeugt, besitzt aber noch nicht alle nötigen Betriebsmittel
  - **Bereit** *(Ready)*
    Prozess besitzt alle nötigen Betriebsmittel und ist bereit zum Laufen
  - **Laufend** *(Running)*
    Prozess wird vom realen Prozessor ausgeführt
  - **Blockiert** *(Blocked/Waiting)*
    Prozess wartet auf ein Ereignis (z.B. Fertigstellung einer Ein- oder Ausgabeoperation, Zuteilung eines Betriebsmittels, Empfang einer Nachricht); zum Warten wird er blockiert
  - **Beendet** *(Terminated)*
    Prozess ist beendet; einige Betriebsmittel sind jedoch noch nicht freigegeben oder Prozess muss aus anderen Gründen im System verbleiben

2.1 Prozesszustände (2)

- Zustandsdiagramm

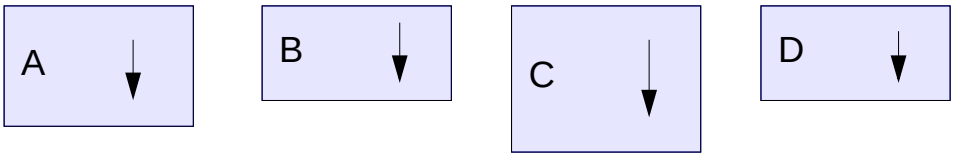


Nach Silberschatz, 1994

- Scheduler ist der Teil des Betriebssystems, der die Zuteilung des realen Prozessors vornimmt.

2.2 Prozesswechsel

- Konzeptionelles Modell

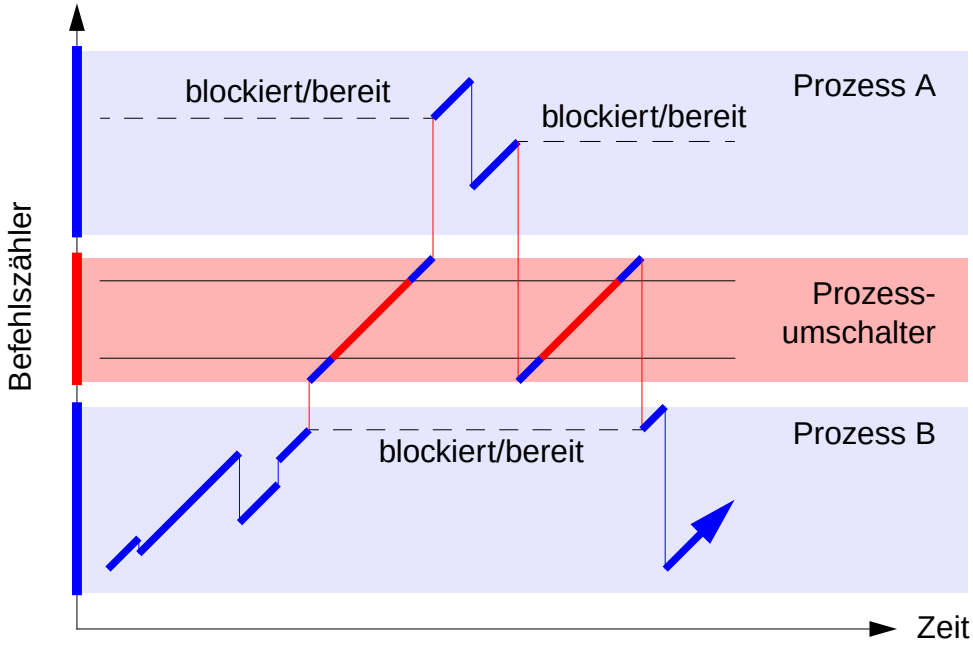


vier Prozesse mit eigenständigen Befehlszählern

- Umschaltung *(Context Switch)*
  - Sichern der Register des laufenden Prozesses inkl. Programmzähler (Kontext),
  - Auswahl des neuen Prozesses,
  - Ablaufumgebung des neuen Prozesses herstellen (z.B. Speicherabbildung, etc.),
  - gesicherte Register laden und
  - Prozessor aufsetzen.

2.2 Prozesswechsel (2)

- Umschaltung



2.2 Prozesswechsel (3)

- Prozesskontrollblock *(Process Control Block; PCB)*
  - Datenstruktur, die alle nötigen Daten für einen Prozess hält. Beispielsweise in UNIX:
    - Prozessnummer *(PID)*
    - verbrauchte Rechenzeit
    - Erzeugungszeitpunkt
    - Kontext (Register etc.)
    - Speicherabbildung
    - Eigentümer *(UID, GID)*
    - Wurzelkatalog, aktueller Katalog
    - offene Dateien
    - ...

2.2 Prozesswechsel (4)

- Prozesswechsel unter Kontrolle des Betriebssystems
  - Mögliche Eingriffspunkte:
    - Systemaufrufe
    - Unterbrechungen
  - Wechsel nach/in Systemaufrufen
    - Warten auf Ereignisse (z.B. Zeitpunkt, Nachricht, Lesen eines Plattenblock)
    - Terminieren des Prozesses
  - Wechsel nach Unterbrechungen
    - Ablauf einer Zeitscheibe
    - bevorzugter Prozess wurde laufbereit
- Auswahlstrategie zur Wahl des nächsten Prozesses
  - *Scheduler*-Komponente

2.3 Prozesserzeugung (UNIX)

- Erzeugen eines neuen UNIX-Prozesses
  - Duplizieren des gerade laufenden Prozesses

```
pid_t fork( void );
```

Vater

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```

Kind

```
pid_t p;
...
p= fork();
if( p == (pid_t)0 ) {
    /* child */
    ...
} else if( p!=(pid_t)-1 ) {
    /* parent */
    ...
} else {
    /* error */
    ...
}
```


2.3 Prozesserzeugung (2)

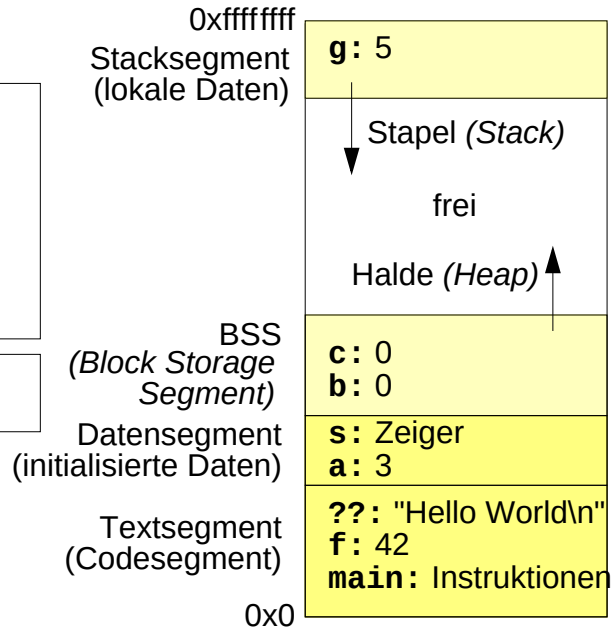
- Der Kind-Prozess ist eine perfekte Kopie des Vaters
  - Gleiches Programm
  - Gleiche Daten (gleiche Werte in Variablen)
  - Gleicher Programmzähler (nach der Kopie)
  - Gleicher Eigentümer
  - Gleiches aktuelles Verzeichnis
  - Gleiche Dateien geöffnet (selbst Schreib-, Lesezeiger ist gemeinsam)
  - ...
- Unterschiede:
  - Verschiedene PIDs
  - **`fork()`** liefert verschiedene Werte als Ergebnis für Vater und Kind

2.4 Speicheraufbau eines Prozesses (UNIX)

- Aufteilung des Hauptspeichers eines Prozesses in Segmente

```
int a= 3, b, c= 0;
const int f= 42;
const char *s= "Hello World\n";

int main( ... ) {
    int g= 5;
}
```

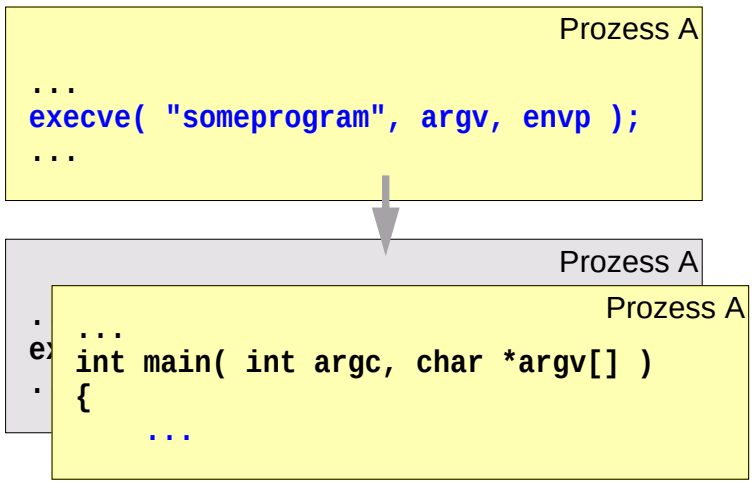
```
s[1]= 'a';
f= 2;
```



2.5 Ausführen eines Programms (UNIX)

- Prozess führt ein neues Programm aus

```
int execve( const char *path, char *const argv[],
            char *const envp[] );
```



Altes ausgeführtes Programm ist endgültig beendet.

2.6 Operationen auf Prozessen (UNIX)

◆ Prozess beenden

```
void _exit( int status );
[ void exit( int status ); ]
```

◆ Prozessidentifikator

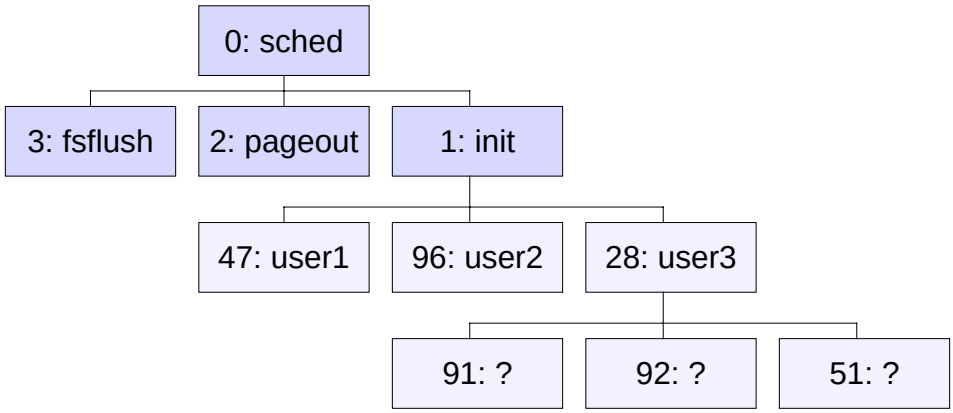
```
pid_t getpid( void );       /* eigene PID */
pid_t getppid( void );      /* PID des Vaterprozesses */
```

◆ Warten auf Beendigung eines Kindprozesses

```
pid_t wait( int *statusp );
```


2.7 Prozesshierarchie (Solaris)

■ Hierarchie wird durch Vater-Kind–Beziehung erzeugt



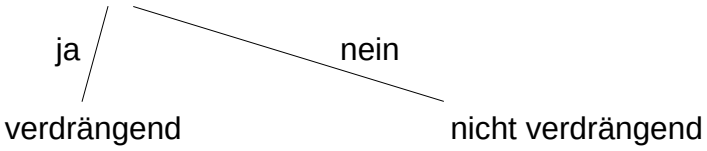
Frei nach Silberschatz 1994

◆ Nur der Vater kann auf das Kind warten

◆ Init-Prozess adoptiert verwaiste Kinder

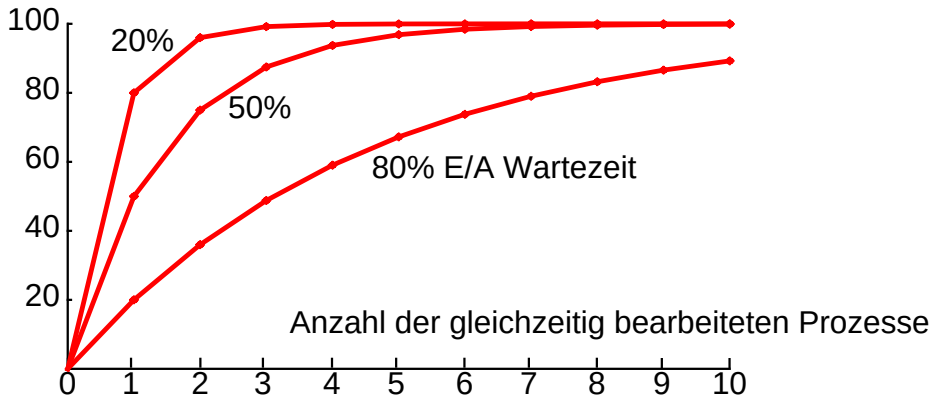
3 Auswahlstrategien

- Strategien zur Auswahl des nächsten Prozesses *(Scheduling Strategies)*
  - Mögliche Stellen zum Treffen von Scheduling-Entscheidungen
    1. Prozess wechselt vom Zustand „laufend“ zum Zustand „blockiert“ (z.B. Ein-, Ausgabeoperation)
    2. Prozess wechselt von „laufend“ nach „bereit“ (z.B. bei einer Unterbrechung des Prozessors)
    3. Prozess wechselt von „blockiert“ nach „bereit“
    4. Prozess terminiert
  - Keine Wahl bei 1. und 4.
  - Wahl bei 2. und 3.



3 Auswahlstrategien (2)

- CPU Auslastung
  - CPU soll möglichst vollständig ausgelastet sein
- ★ CPU-Nutzung in Prozent, abhängig von der Anzahl der Prozesse und deren prozentualer Wartezeit



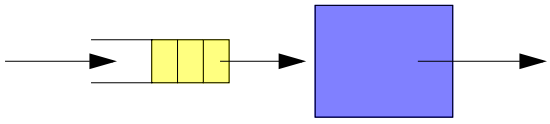
Nach Tanenbaum, 1995

3 Auswahlstrategien (3)

- Durchsatz
  - Möglichst hohe Anzahl bearbeiteter Prozesse pro Zeiteinheit
- Verweilzeit
  - Gesamtzeit des Prozesses in der Rechenanlage soll so gering wie möglich sein
- Wartezeit
  - Möglichst kurze Gesamtzeit, in der der Prozess im Zustand „bereit“ ist
- Antwortzeit
  - Möglichst kurze Reaktionszeit des Prozesses im interaktiven Betrieb

3.1 First Come, First Served

- Der erste Prozess wird zuerst bearbeitet *(FCFS)*
  - „Wer zuerst kommt ...“
  - Nicht-verdrängend
- Warteschlange zum Zustand „bereit“
  - Prozesse werden hinten eingereiht
  - Prozesse werden vorne entnommen



- Bewertung
  - fair (?)
  - Wartezeiten nicht minimal
  - nicht für Time-Sharing-Betrieb geeignet

3.1 First Come, First Served (2)

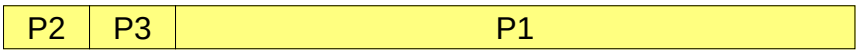
- Beispiel zur Betrachtung der Wartezeiten

Prozess 1:	24	} Zeiteinheiten
Prozess 2:	3	
Prozess 3:	3	

  - Reihenfolge: P1, P2, P3

mittlere Wartezeit: (0+24+27)/3 = 17

  - Reihenfolge: P2, P3, P1



mittlere Wartezeit: (6+0+3)/3 = 3

3.2 Shortest Job First

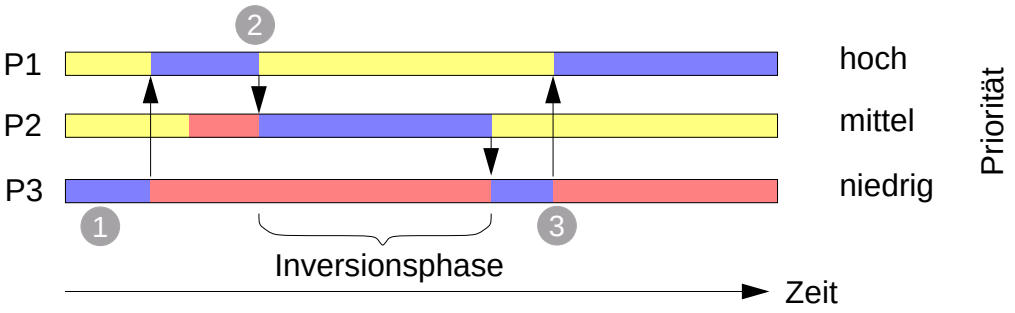
- Kürzester Job wird ausgewählt *(SJF)*
  - Länge bezieht sich auf die nächste Rechenphase bis zur nächsten Warteoperation (z.B. Ein-, Ausgabe)
- „bereit“-Warteschlange wird nach Länge der nächsten Rechenphase sortiert
  - Vorhersage der Länge durch Protokollieren der Länge bisheriger Rechenphasen (Mittelwert, exponentielle Glättung)
  - ... Protokollierung der Länge der vorherigen Rechenphase
- SJF optimiert die mittlere Wartezeit
  - Da Länge der Rechenphase in der Regel nicht genau vorhersagbar, nicht ganz optimal.
- Varianten: verdrängend *(PSJF)* und nicht-verdrängend

3.3 Prioritäten

- Prozess mit höchster Priorität wird ausgewählt
  - dynamisch — statisch
    (z.B. SJF: dynamische Vergabe von Prioritäten gemäß Länge der nächsten Rechenphase)
    (z.B. statische Prioritäten in Echtzeitsystemen; Vorhersagbarkeit von Reaktionszeiten)
  - verdrängend — nicht-verdrängend
- Probleme
  - Aushungerung
    Ein Prozess kommt nie zum Zuge, da immer andere mit höherer Priorität vorhanden sind.
  - Prioritätenumkehr *(Priority Inversion)*

3.3 Prioritäten (2)

- Prioritätenumkehr
  - hochpriorer Prozess wartet auf ein Betriebsmittel, das ein niedrigpriorer Prozess besitzt; dieser wiederum wird durch einen mittelprioren Prozess verdrängt und kann daher das Betriebsmittel gar nicht freigeben



laufend
bereit
blockiert

1. P3 fordert Betriebsmittel an
2. P1 wartet auf das gleiche Betriebsmittel
3. P3 gibt Betriebsmittel frei

3.3 Prioritäten (3)

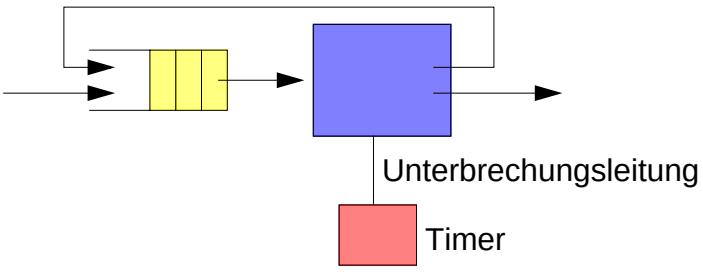
★ Lösungen

- ◆ zur Prioritätenumkehr:
  dynamische Anhebung der Priorität für kritische Prozesse
- ◆ zur Aushungerung:
  dynamische Anhebung der Priorität für lange wartende Prozesse
  (Alterung, *Aging*)

3.4 Round-Robin Scheduling

■ Zuteilung und Auswahl erfolgt reihum

- ◆ ähnlich FCFS aber mit Verdrängung
- ◆ Zeitquant *(Time Quantum)* oder Zeitscheibe *(Time Slice)* wird zugeteilt
- ◆ geeignet für *Time-Sharing*-Betrieb



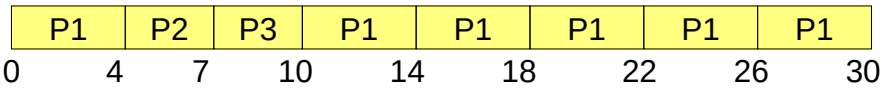
- ◆ Wartezeit ist jedoch eventuell relativ lang

3.4 Round-Robin Scheduling (2)

- Beispiel zur Betrachtung der Wartezeiten

Prozess 1:	24	} Zeiteinheiten
Prozess 2:	3	
Prozess 3:	3	

  - Zeitquant ist 4 Zeiteinheiten
  - Reihenfolge in der „bereit“-Warteschlange: P1, P2, P3



0 4 7 10 14 18 22 26 30

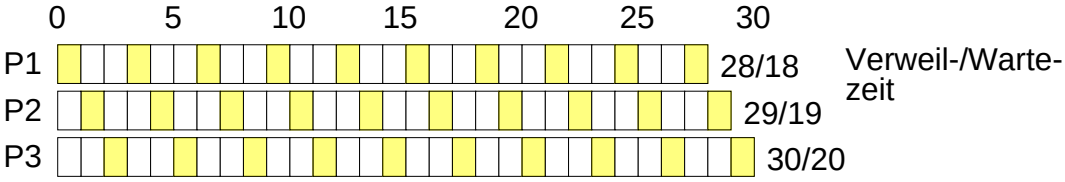
mittlere Wartezeit: (6+4+7)/3 = 5.7

3.4 Round-Robin Scheduling (3)

- Effizienz hängt von der Größe der Zeitscheibe ab
  - kurze Zeitscheiben: Zeit zum Kontextwechsel wird dominant
  - lange Zeitscheiben: Round Robin nähert sich FCFS an
- Verweilzeit und Wartezeit hängt ebenfalls von der Zeitscheibengröße ab
  - Beispiel: 3 Prozesse mit je 10 Zeiteinheiten Rechenbedarf
    - Zeitscheibengröße 1
    - Zeitscheibengröße 10

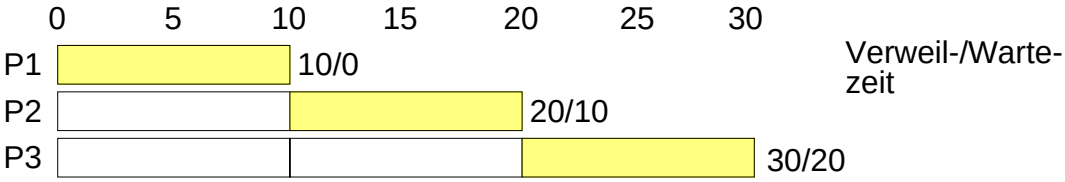
3.4 Round-Robin Scheduling (4)

- Zeitscheibengröße 1:



durchschnittliche Verweilzeit: 29 Zeiteinheiten = (28+29+30)/3
durchschnittliche Wartezeit: 19 Zeiteinheiten = (18+19+20)/3

- Zeitscheibengröße 10:



durchschnittliche Verweilzeit: 20 Zeiteinheiten = (10+20+30)/3
durchschnittliche Wartezeit: 10 Zeiteinheiten = (0+10+20)/3

3.5 Multilevel-Queue Scheduling

- Verschiedene Schedulingklassen
  - z.B. Hintergrundprozesse (Batch) und Vordergrundprozesse (interaktive Prozesse)
  - jede Klasse besitzt ihre eigenen Warteschlangen und verwaltet diese nach einem eigenen Algorithmus
  - zwischen den Klassen gibt es ebenfalls ein Schedulingalgorithmus z.B. feste Prioritäten (Vordergrundprozesse immer vor Hintergrundprozessen)
- Beispiel: Solaris
  - Schedulingklassen
    - Systemprozesse
    - Real-Time Prozesse
    - Time-Sharing Prozesse
    - interaktive Prozesse

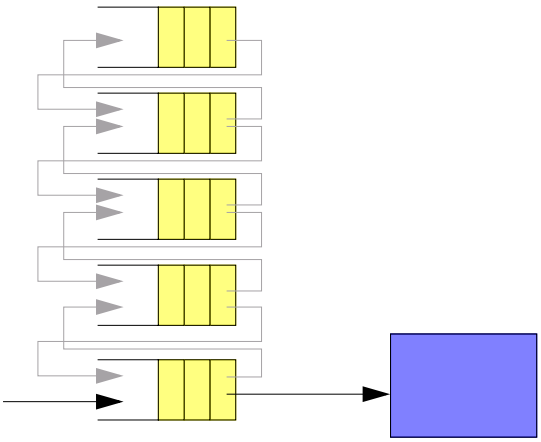
3.5 Multilevel-Queue Scheduling (2)

- ◆ Scheduling zwischen den Klassen mit fester Priorität
  (z.B. Real-Time-Prozesse vor Time-Sharing-Prozessen)
- ◆ In jeder Klasse wird ein eigener Algorithmus benutzt:
  - Systemprozesse: FCFS
  - Real-Time Prozesse: statische Prioritäten
  - Time-Sharing und interaktive Prozesse:
    ausgefeiltes Verfahren zur Sicherung von:
    - kurzen Reaktionszeiten
    - fairer Zeitaufteilung zwischen rechenintensiven und I/O-intensiven Prozessen
    - gewisser Benutzersteuerung

★ Multilevel Feedback Queue Scheduling

3.6 Multilevel-Feedback-Queue Scheduling

- ■ Mehrere Warteschlangen *(MLFB)*
  - ◆ jede Warteschlange mit eigener Behandlung
  - ◆ Prozesse können von einer zur anderen Warteschlange transferiert werden



3.6 Multilevel Feedback Queue Scheduling (2)

- Beispiel:
  - mehrere Warteschlangen mit Prioritäten (wie bei Multilevel Queue)
  - Prozesse, die lange rechnen, wandern langsam in Warteschlangen mit niedrigerer Priorität (bevorzugt interaktive Prozesse)
  - Prozesse, die lange warten müssen, wandern langsam wieder in höherpriore Warteschlangen *(Aging)*

3.7 Beispiel: Time Sharing Scheduling in Solaris

- 60 Warteschlangen, Tabellensteuerung

Level	ts_quantum	ts_tqexp	ts_maxwait	ts_lwait	ts_slpret
0	200	0	0	50	50
1	200	0	0	50	50
2	200	0	0	50	50
3	200	0	0	50	50
4	200	0	0	50	50
5	200	0	0	50	50
...					
44	40	34	0	55	55
45	40	35	0	56	56
46	40	36	0	57	57
47	40	37	0	58	58
48	40	38	0	58	58
49	40	39	0	59	58
50	40	40	0	59	58
51	40	41	0	59	58
52	40	42	0	59	58
53	40	43	0	59	58
54	40	44	0	59	58
55	40	45	0	59	58
56	40	46	0	59	58
57	40	47	0	59	58
58	40	48	0	59	58
59	20	49	32000	59	59

3.7 Beispiel: TS Scheduling in Solaris (2)

- Tabelleninhalt
  - kann ausgelesen und gesetzt werden (Auslesen: **`dispadmin -c TS -g`**)
  - **`Level:`** Nummer der Warteschlange Hohe Nummer = hohe Priorität
  - **`ts_quantum:`** maximale Zeitscheibe für den Prozess (in Millisek.)
  - **`ts_tqexp:`** Warteschlangennummer, falls der Prozess die Zeitscheibe aufbraucht
  - **`ts_maxwait:`** maximale Zeit für den Prozess in der Warteschlange ohne Bedienung (in Sekunden; Minimum ist eine Sekunde)
  - **`ts_lwait:`** Warteschlangennummer, falls Prozess zulange in dieser Schlange
  - **`ts_slpret:`** Warteschlangenummer für das Wiedereinreihen nach einer blockierenden Aktion

3.7 Beispiel: TS Scheduling in Solaris (3)

- Beispielprozess:
  - 1000ms Rechnen am Stück
  - 5 E/A Operationen mit jeweils Rechenzeiten von 1ms dazwischen

#	Warteschlange	Rechenzeit	Prozesswechsel weil ...
1	59	20	Zeitquant abgelaufen
2	49	40	Zeitquant abgelaufen
3	39	80	Zeitquant abgelaufen
4	29	120	Zeitquant abgelaufen
5	19	160	Zeitquant abgelaufen
6	9	200	Zeitquant abgelaufen
7	0	200	Zeitquant abgelaufen
8	0	180	E/A Operation
9	50	1	E/A Operation
10	58	1	E/A Operation
11	58	1	E/A Operation
12	58	1	E/A Operation

3.7 Beispiel: TS Scheduling in Solaris (4)

- Tabelle gilt nur unter der folgenden Bedingung:
  - ◆ Prozess läuft fast alleine, andernfalls
    - könnte er durch höherpriore Prozesse verdrängt und/oder ausgebremst werden,
    - wird er bei langem Warten in der Priorität wieder angehoben.
- Beispiel:

#	Warteschlange	Rechenzeit	Prozesswechsel weil ...
		...	
6	9	200	Zeitquant abgelaufen
7	0	20	Wartezeit von 1s abgelaufen
8	50	40	Zeitquant abgelaufen
9	40	40	Zeitquant abgelaufen
10	30	80	Zeitquant abgelaufen
11	20	120	Zeitquant abgelaufen
		...	

3.7 Beispiel: TS Scheduling in Solaris (5)

- Weitere Einflussmöglichkeiten
  - ◆ Anwender und Administratoren können Prioritätenoffsets vergeben
  - ◆ Die Offsets werden auf die Tabellenwerte addiert und ergeben die wirklich verwendete Warteschlange
  - ◆ positive Offsets: Prozess wird bevorzugt
  - ◆ negative Offsets: Prozess wird benachteiligt
  - ◆ Außerdem können obere Schranken angegeben werden
- Systemaufruf
  - ◆ Verändern der eigenen Prozesspriorität

```
int nice( int incr );
```

(positives Inkrement: niedrigere Priorität;
negatives Inkrement: höhere Priorität)

4 Prozesskommunikation

- *Inter-Process-Communication (IPC)*
  - ◆ Mehrere Prozesse bearbeiten eine Aufgabe
    - gleichzeitige Nutzung von zur Verfügung stehender Information durch mehrere Prozesse
    - Verkürzung der Bearbeitungszeit durch Parallelisierung
- Kommunikation durch Nachrichten
  - ◆ Nachrichten werden zwischen Prozessen ausgetauscht
- Kommunikation durch gemeinsamen Speicher
  - ◆ F. Hofmann nennt dies Kooperation (kooperierende Prozesse)

4 Prozesskommunikation (2)

- Klassifikation nachrichtenbasierter Kommunikation
  - ◆ Klassen
    - Kanäle *(Pipes)*
    - Kommunikationsendpunkte *(Sockets, Ports)*
    - Briefkästen, Nachrichtenpuffer *(Queues)*
    - Unterbrechungen *(Signals)*
  - ◆ Übertragunsrichtung
    - unidirektional
    - bidirektional (voll-duplex, halb-duplex)

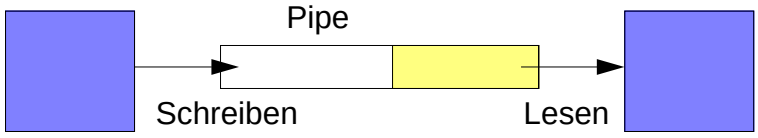
4 Prozesskommunikation (3)

◆ Übertragungs- und Aufrufeigenschaften
- zuverlässig — unzuverlässig
- gepuffert — ungepuffert
- blockierend — nichtblockierend
- stromorientiert — nachrichtenorientiert — RPC

◆ Adressierung
- implizit: UNIX Pipes
- explizit: Sockets
- globale Adressierung: Sockets, Ports
- Gruppenadressierung: Multicast, Broadcast
- funktionale Adressierung: Dienste

4.1 Pipes

■ Kanal zwischen zwei Kommunikationspartnern
◆ unidirektional
(heute gleichzeitige Erzeugung zweier Pipes je eine pro Richtung)
◆ gepuffert (feste Puffergröße), zuverlässig, stromorientiert



■ Operationen: Schreiben und Lesen
◆ Ordnung der Zeichen bleibt erhalten (Zeichenstrom)
◆ Blockierung bei voller Pipe (Schreiben) und leerer Pipe (Lesen)

4.1 Pipes (2)

- Systemaufruf unter Solaris
  - Öffnen einer Pipe

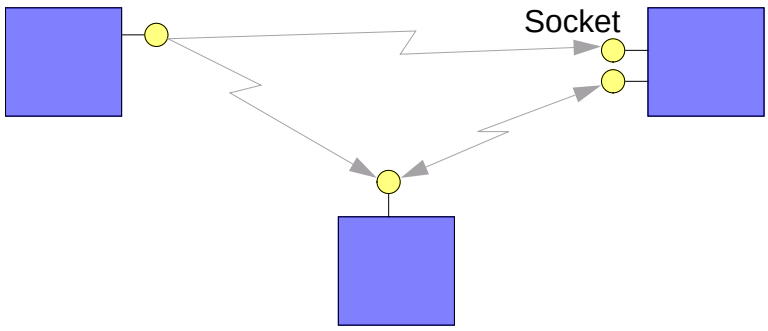
    ```
    int pipe( int fdes[2] );
    ```

  - Es werden eigentlich zwei Pipes geöffnet
    `fdes[0]` liest aus Pipe 1 und schreibt in Pipe 2
    `fdes[1]` liest aus Pipe 2 und schreibt in Pipe 1

  - Zugriff auf Pipes wie auf eine Datei: **`read`** und **`write`**, **`readv`** und **`writev`**
- Named-Pipes
  - Pipes können auch als Spezialdateien ins Dateisystem gelegt werden.
  - Standardfunktionen zum Lesen und Schreiben können dann verwendet werden.

4.2 Sockets

- Allgemeine Kommunikationsendpunkte
  - bidirektional, gepuffert



  - Auswahl einer Protokollfamilie
    - z.B. Internet (TCP/IP), UNIX (innerhalb von Prozessen der gleichen Maschine), ISO, Appletalk, DECnet, SNA, ...
    - durch die Protokollfamilie wird gleichzeitig die Adressfamilie festgelegt (Struktur zur Bezeichnung von Protokolladressen)

4.2 Sockets (2)

◆ Auswahl eines Sockettyps für Protokolle mit folgenden Eigenschaften:
- stromorientiert, verbindungsorientiert und gesichert
- nachrichtenorientiert und ungesichert (Datagramm)
- nachrichtenorientiert und gesichert

◆ Auswahl eines Protokolls der Familie
- z.B. UDP

◆ explizite Adressierung
- Unicast: genau ein Kommunikationspartner
- Multicast: eine Gruppe
- Broadcast: alle möglichen Adressaten

◆ Sockets können blockierend und nichtblockierend betrieben werden.

4.2 Sockets (3)

■ UNIX-Domain

◆ UNIX-Domain-Sockets verhalten sich wie bidirektionale Pipes.

◆ Anlage als Spezialdatei im Dateisystem möglich

■ Internet-Domain

◆ Protokolle:
- TCP/IP (strom- und verbindungsorientiert, gesichert)
- UDP/IP (nachrichtenorientiert, verbindungslos, ungesichert)
  – Nachrichten können verloren oder dupliziert werden
  – Reihenfolge kann durcheinander geraten
  – Paketgrenzen bleiben erhalten
  (Datagramm-Protokoll)

◆ Adressen: IP-Adressen und Port-Nummern

4.2 Sockets (4)

- Anlegen von Sockets
  - ◆ Generieren eines Sockets mit (Rückgabewert ist ein Filedeskriptor)

```
int socket( int domain, int type, int proto );
```

  - ◆ Adresszuteilung
    - Sockets werden ohne Adressen generiert
    - Adressenzuteilung erfolgt automatisch oder durch:

```
int bind( int socket, const struct sockaddr *address,
       size_t address_len);
```


4.2 Sockets (5)

- Datagramm-Sockets
  - ◆ kein Verbindungsaufbau notwendig
  - ◆ Datagramm senden

```
ssize_t sendto( int socket, const void *message,
       size_t length, int flags,
       const struct sockaddr *dest_addr, size_t dest_len);
```

  - ◆ Datagramm empfangen

```
ssize_t recvfrom( int socket, void *buffer,
       size_t length, int flags, struct sockaddr *address,
       size_t *address_len);
```

4.2 Sockets (6)

- Stromorientierte Sockets
  - Verbindungsaufbau notwendig
  - *Client* (Benutzer, Benutzerprogramm) will zu einem *Server* (Dienstanbieter) eine Kommunikationsverbindung aufbauen
- Client: Verbindungsaufbau bei stromorientierten Sockets
  - Verbinden des Sockets mit

```
int connect( int socket, const struct sockaddr *address,
       size_t address_len);
```

  - Senden und Empfangen mit **`write`** und **`read`** (**`send`** und **`recv`**)
  - Beenden der Verbindung mit **`close`** (schließt den Socket)

4.2 Sockets (7)

- Server
  - bindet Socket an eine Adresse (sonst nicht zugreifbar)
  - bereitet Socket auf Verbindungsanforderungen vor durch

```
int listen(int s, int backlog);
```

  - akzeptiert einzelne Verbindungsanforderungen durch

```
int accept(int s, struct sockaddr *addr, int *addrlen);
```

    - gibt einen neuen Socket zurück, der mit dem Client verbunden ist
    - blockiert, falls kein Verbindungswunsch vorhanden
  - liest Daten mit **`read`** und führt den angebotenen Dienst aus
  - schickt das Ergebnis mit **`write`** zurück zum Sender
  - schließt den neuen Socket

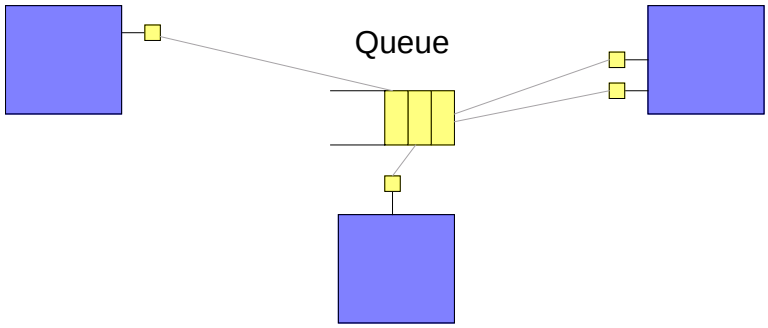
4.3 UNIX Queues

- Nachrichtenpuffer *(Queue, FIFO)*
  - rechnerlokale Adresse *(Key)* dient zur Identifikation eines Puffers
  - prozesslokale Nummer *(MSQID)* ähnlich dem Filedeskriptor (wird bei allen Operationen benötigt)
  - Zugriffsrechte wie auf Dateien
  - ungerichtete Kommunikation, gepuffert (einstellbare Größe pro Queue)
  - Nachrichten haben einen Typ (**`long`**-Wert)
  - Operationen zum Senden und Empfangen einer Nachricht
  - blockierend — nichtblockierend
  - alle Nachrichten — nur ein bestimmter Typ

4.3 UNIX Queues (2)

- Systemaufrufe unter Solaris 2.5
  - Erzeugen einer Queue bzw. Holen einer MSQID

```
int msgget( key_t key, int msgflg );
```



  - Alle kommunizierenden Prozesse müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Queue mit gleichem Key erzeugt werden

4.3 UNIX Queues (3)

- Es können Queues ohne Key erzeugt werden (private Queues)
  - Nicht-private Queues sind persistent
  - Sie müssen explizit gelöscht werden

```
int msgctl( int msqid, int cmd, struct msqid_ds *buf );
```

- Systemkommandos zum Behandeln von Queues
  - Listen aktiver Message-Queues

```
ipcs -q
```

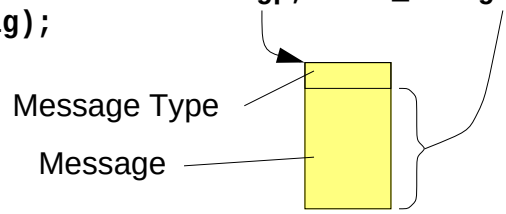
  - Löschen von Queues

```
ipcrm -Q <key>
```


4.3 UNIX Queues (4)

- Operationen auf Queues
  - Senden einer Nachricht

```
int msgsnd( int msqid, const void *msgp, size_t msgsz,
            int msgflg);
```



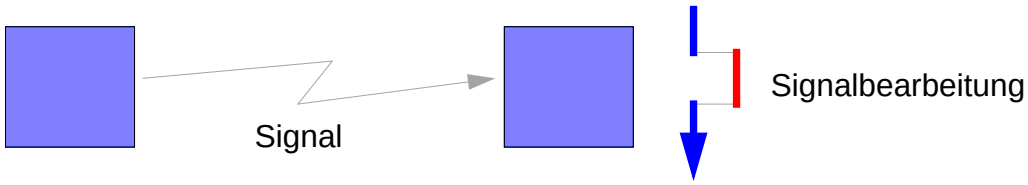
  - Empfangen einer Nachricht

```
int msgrcv( int msqid, void *msgp, size_t msgsz,
            long msgtype, int msgflg);
```

  - Zugriffsrechte werden beachtet

4.4 UNIX Signale

- Signale sind Unterbrechungen ähnlich denen eines Prozessors
  - ◆ Prozess führt eine definierte Signalbehandlung durch
    - Ignorieren
    - Terminierung des Prozesses
    - Aufruf einer Funktion
  - ◆ Nach der Behandlung läuft Prozess an unterbrochener Stelle weiter



4.4 UNIX Signale (2)

- Kommunikation über Signale: Signalisierung von Ereignissen

Signale (Beispiele)		Voreingestelltes Verhalten
◆ Terminaleingabe		
• **SIGINT**		
Interrupt	^C	*Prozess terminiert*
• **SIGQUIT**		
Quit	^|	*Prozess terminiert, Core dump*
◆ Systemsignale ausgelöst durch den Prozess selbst		
• **SIGBUS**		
Bus error		*Prozess terminiert, Core dump*
• **SIGSEGV**		
Segmentation fault		*Prozess terminiert, Core dump*

4.4 UNIX Signale (3)

Signale (Beispiele) — Voreingestelltes Verhalten

◆ Systemsignale ausgelöst durch Betriebssystem

- **SIGALRM**
  Alarmzeitgeber — *Prozess terminiert*
- **SIGCHLD**
  Kindprozessstatus — *wird ignoriert*

◆ Benutzerdefinierte Signale

- **SIGUSR1, SIGUSR2**
  frei für Benutzerkommunikation, z.B. für Start und Ende einer Bearbeitung — *Prozess terminiert*

4.4 UNIX Signale (4)

■ Signalbehandlung kann eingestellt werden:

◆ SIG_IGN: Ignorieren des Signals

◆ SIG_DFL: Defaultverhalten einstellen

◆ *Funktionsadresse:* Funktion wird in der Signalbehandlung aufgerufen und ausgeführt

■ UNIX Systemaufrufe

◆ Einfangen von Signalen

```
void (*signal( int sig, void (*disp)( int ) ))( int );
```

◆ Zustellen von Signalen

```
int kill( pid_t pid, int sig );
```

4.4 UNIX Signale (5)

- Signalsemantik unterschiedlich bei verschiedenen UNIX Systemen

- **BSD, Posix:**
  Blockieren weiterer Signale während der Behandlung
  - Beim Einfangen werden weitere gleichartige Signale blockiert (maximal wird ein Signal gespeichert).
  - Sobald die Behandlung fertig ist, wird die Blockierung wieder freigegeben.

- **System V:**
  Rücksetzen der Signalbehandlung beim Einfangen eines Signals
  - Beim Einfangen eines Signals wird implizit **`signal( ..., SIG_DFL)`** aufgerufen.
  - Im Signalhandler muss der Handler selbst wieder eingesetzt werden.
  - kurze Zeitspanne ohne Signalhandler

4.4 UNIX Signale (5)

- **System VR4:**
  Unterbrechung von Systemaufrufen
  - Fast alle „langsamen“ Systemaufrufe können durch die Signalbehandlung unterbrochen werden.
  - **`errno`** wird auf **`EINTR`** gesetzt und der Systemaufruf terminiert mit **`-1`**.
  - Wenn kein automatischer Wiederanlauf nach einer Unterbrechung durchgeführt wird, muss der Anwender auf den Fehler **`EINTR`** reagieren.

```
...
cnt= write( fd, buf, 100 );
...
```

```
...
do {
    cnt= write( fd, buf, 100 );
}
while( cnt < 0 && errno == EINTR );
...
```

4.4 UNIX Signale (6)

- Moderne UNIX Systeme implementieren alle Variationen
  - Systemaufruf statt **`signal`**:

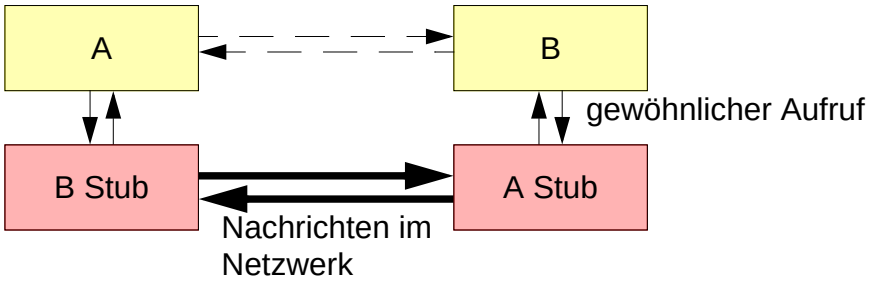
```
int sigaction( int sig, const struct sigaction *act,
               struct sigaction *oact );
```

  - Rücksetzen auf Defaulthandler einstellbar
  - Liste von Signalen einstellbar, die beim Einfangen eines Signals blockiert werden soll
  - Automatischer Wiederanlauf von unterbrochenen Systemaufrufen einstellbar

★ **Wichtig:** Sie müssen die Semantik der Signalbehandlung auf dem entsprechenden UNIX System kennen!

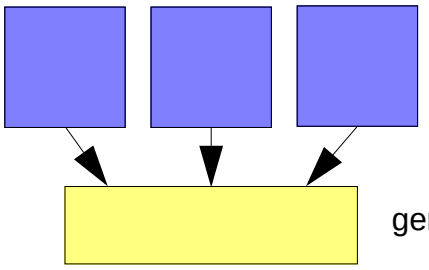
4.5 Fernaufruf *(RPC)*

- Funktionsaufruf über Prozessgrenzen hinweg *(Remote procedure call)*
  - hoher Abstraktionsgrad
  - selten wird Fernaufruf direkt vom System angeboten; benötigt Abbildung auf andere Kommunikationsformen z.B. auf Nachrichten
  - Abbildung auf mehrere Nachrichten
    - Auftragsnachricht transportiert Aufrufabsicht und Parameter.
    - Ergebnisnachricht transportiert Ergebnisse des Aufrufs.



4.6 Gemeinsamer Speicher

- Zwei Prozesse können auf einen gemeinsamen Speicherbereich zugreifen
  - gemeinsame Variablen oder Datenstrukturen



- Einrichten von gemeinsamem Speicher erst im Abschnitt E.5.

5 Aktivitätsträger *(Threads)*

- Mehrere Prozesse zur Strukturierung von Problemlösungen
  - Aufgaben eines Prozesses leichter modellierbar, wenn in mehrere kooperierende Prozesse unterteilt
    z.B. Anwendungen mit mehreren Fenstern (ein Prozess pro Fenster)
    z.B. Anwendungen mit vielen gleichzeitigen Aufgaben (Webbrowser)
  - Multiprozessorsysteme werden erst mit mehreren parallel laufenden Prozessen ausgenutzt
    z.B. wissenschaftliches Hochleistungsrechnen (Aerodynamik etc.)
  - Client-Server-Anwendungen unter UNIX: pro Anfrage wird ein neuer Prozess gestartet
    z.B. Webserver

5.1 Prozesse mit gemeinsamem Speicher

■ Gemeinsame Nutzung von Speicherbereichen durch mehrere Prozesse

▲ Nachteile

◆ viele Betriebsmittel zur Verwaltung eines Prozesses notwendig
- Dateideskriptoren
- Speicherabbildung
- Prozesskontrollblock

◆ Prozessumschaltungen sind aufwendig.

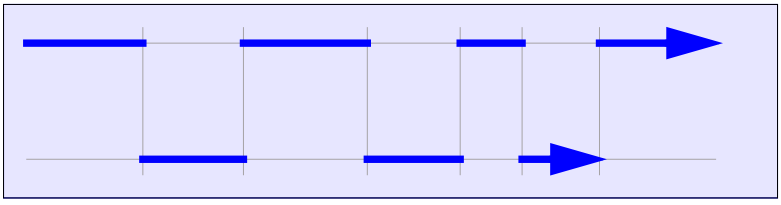
★ Vorteil

◆ In Multiprozessorsystemen sind echt parallele Abläufe möglich.

5.2 Koroutinen

■ Einsatz von Koroutinen

◆ einige Anwendungen lassen sich mit Hilfe von Koroutinen (auf Benutzerebene) innerhalb eines Prozesses gut realisieren



ein Prozess
zwei Koroutinen

▲ Nachteile:

◆ Scheduling zwischen den Koroutinen schwierig (Verdrängung meist nicht möglich)

◆ in Multiprozessorsystemen keine parallelen Abläufe möglich

◆ Wird eine Koroutine in einem Systemaufruf blockiert, ist der gesamte Prozess blockiert.

5.3 Aktivitätsträger

★ **Lösungsansatz:**
Aktivitätsträger *(Threads)* oder
leichtgewichtige Prozesse *(Lightweighted Processes, LWPs)*

- ◆ Eine Gruppe von Threads nutzt gemeinsam eine Menge von Betriebsmitteln.
  - Instruktionen
  - Datenbereiche
  - Dateien, Semaphoren etc.
- ◆ Jeder Thread repräsentiert eine eigene Aktivität:
  - eigener Programmzähler
  - eigener Registersatz
  - eigener Stack

5.3 Aktivitätsträger (2)

- ◆ Umschalten zwischen zwei Threads einer Gruppe ist erheblich billiger als eine normale Prozessumschaltung.
  - Es müssen nur die Register und der Programmzähler gewechselt werden (entspricht dem Aufwand für einen Funktionsaufruf).
  - Speicherabbildung muss nicht gewechselt werden.
  - Alle Systemressourcen bleiben verfügbar.

■ Ein UNIX-Prozess ist ein Adressraum mit einem Thread

- ◆ Solaris: Prozess kann mehrere Threads besitzen

■ Implementierungen von Threads

- ◆ User-level Threads
- ◆ Kernel-level Threads

5.4 User-Level-Threads

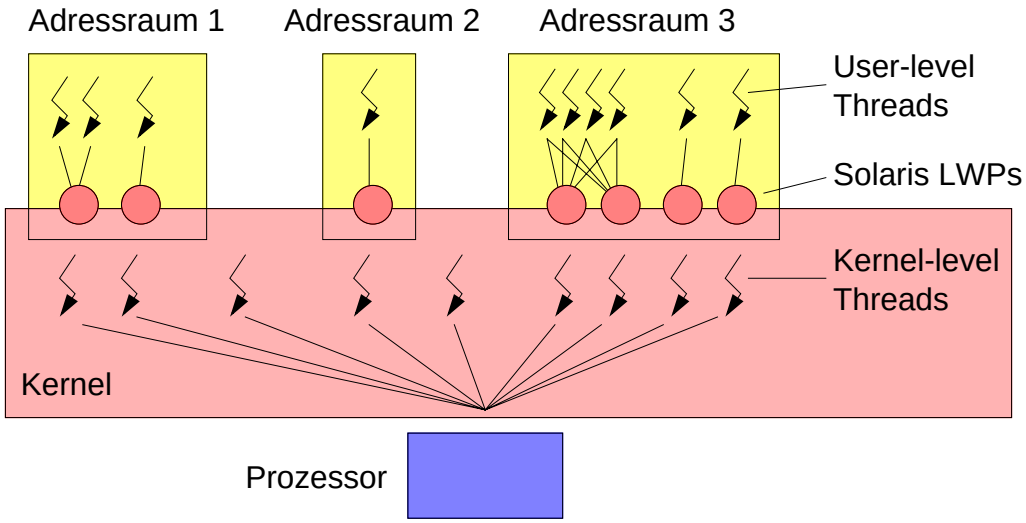
- Implementierung
  - Instruktionen im Anwendungsprogramm schalten zwischen den Threads hin- und her (ähnlich wie der Scheduler im Betriebssystem)
  - Betriebssystem sieht nur einen Thread
- ★ Vorteile
  - keine Systemaufrufe zum Umschalten erforderlich
  - effiziente Umschaltung
  - Schedulingstrategie in der Hand des Anwenders
- ▲ Nachteile
  - Bei blockierenden Systemaufrufen bleiben alle User-Level-Threads stehen.
  - Kein Ausnutzen eines Multiprozessors möglich

5.5 Kernel-Level-Threads

- Implementierung
  - Betriebssystem kennt Kernel-Level-Threads
  - Betriebssystem schaltet Threads um
- ★ Vorteile
  - kein Blockieren unbeteiligter Threads bei blockierenden Systemaufrufen
- ▲ Nachteile
  - weniger effizientes Umschalten
  - Fairnessverhalten nötig
    (zwischen Prozessen mit vielen und solchen mit wenigen Threads)
  - Schedulingstrategie meist vorgegeben

5.6 Beispiel: LWPs und Threads (Solaris)

- Solaris kennt Kernel-, User-Level-Threads und LWPs



Nach Silberschatz, 1994

6 Koordinierung

- Beispiel: Beobachter und Protokollierer
  - Mittels Induktionsschleife werden Fahrzeuge gezählt. Alle 10min druckt der Protokollierer die im letzten Zeitraum vorbeigekommene Anzahl aus.

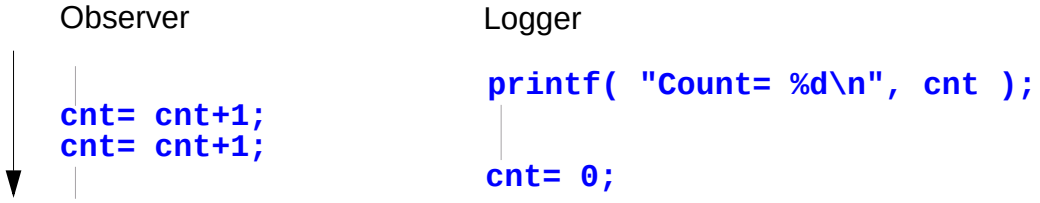
```
int cnt= 0;

Observer
    on indication
        cnt= cnt+1;

Logger
    every 10 minutes
        printf( "Count= %d\n", cnt );
        cnt= 0;
```

6 Koordinierung (2)

- Effekte:
  - Fahrzeuge gehen „verloren“
  - Fahrzeuge werden doppelt gezählt
- Ursachen:
  - Befehle in C werden nicht unteilbar (atomar) abgearbeitet, da sie auf mehrere Maschinenbefehle abgebildet werden.
  - In C werden keinesfalls mehrere Anweisungen zusammen atomar abgearbeitet.
  - Prozesswechsel innerhalb einer Anweisung oder zwischen zwei zusammengehörigen Anweisungen können zu Inkonsistenzen führen.

6 Koordinierung (3)

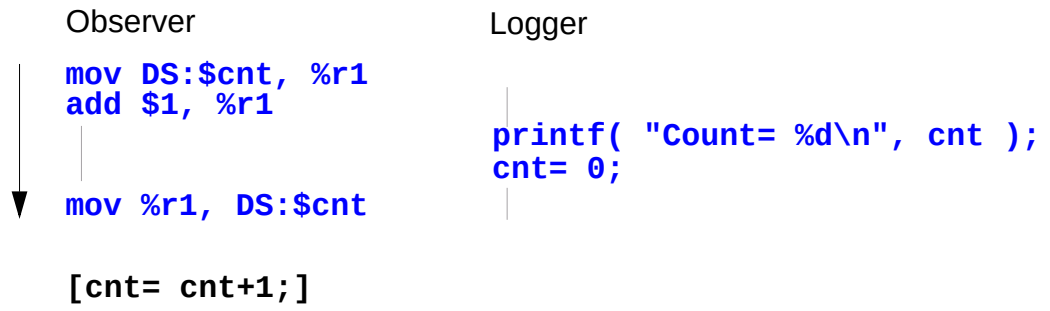
- Fahrzeuge gehen „verloren“
  - Nach dem Drucken wird der Protokollierer unterbrochen. Beobachter zählt weitere Fahrzeuge. Anzahl wird danach ohne Beachtung vom Protokollierer auf Null gesetzt.



6 Koordinierung (4)

▲ Fahrzeuge werden doppelt gezählt:

◆ Beobachter will Zähler erhöhen und holt sich diesen dazu in ein Register. Er wird unterbrochen und der Protokollierer setzt Anzahl auf Null. Beobachter erhöht Registerwert und schreibt diesen zurück. Dieser Wert wird erneut vom Protokollierer registriert.



6 Koordinierung (5)

■ Gemeinsame Nutzung von Daten oder Betriebsmitteln

◆ kritische Abschnitte:

- nur einer soll Zugang zu Daten oder Betriebsmitteln haben (gegenseitiger Ausschluss, *Mutual Exclusion*, *Mutex*)
- kritische Abschnitte erscheinen allen anderen als zeitlich unteilbar

◆ Wie kann der gegenseitige Ausschluss in kritischen Abschnitten erzielt werden?

■ Koordinierung allgemein:

◆ Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

★ Hinweis:

◆ Im Folgenden wird immer von Prozessen die Rede sein. Koordinierung kann/muss selbstverständlich auch zwischen Threads stattfinden.

6.1 Gegenseitiger Ausschluss

- Zwei Prozesse wollen regelmäßig kritischen Abschnitt betreten
  - Annahme: Maschinenbefehle sind unteilbar (atomar)
- 1. Versuch

```
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    while( turn == 1 );
    ...
    /* critical sec. */
    ...
    turn= 1;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    while( turn == 0);
    ...
    /* critical sec. */
    ...
    turn= 0;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (2)

- Probleme der Lösung
  - nur alternierendes Betreten des kritischen Abschnitts durch P_0 und P_1 möglich
  - Implementierung ist unvollständig
  - aktives Warten
- Ersetzen von **`turn`** durch zwei Variablen **`ready0`** und **`ready1`**
  - **`ready0`** zeigt an, dass Prozess 0 bereit für den kritischen Abschnitt ist
  - **`ready1`** zeigt an, dass Prozess 1 bereit für den kritischen Abschnitt ist

6.1 Gegenseitiger Ausschluss (3)

- 2. Versuch

```
bool ready0= FALSE;
bool ready1= FALSE;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    while( ready1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    while( ready0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```


6.1 Gegenseitiger Ausschluss (4)

- Gegenseitiger Ausschluss wird erreicht
  - leicht nachweisbar durch Zustände von **`ready0`** und **`ready1`**
- Probleme der Lösung
  - aktives Warten
  - Verklemmung möglich

6.1 Gegenseitiger Ausschluss (5)

- Betrachtung der nebenläufigen Abfolgen

P_0	P_1
`ready0= TRUE;`	`ready1= TRUE;`
`while( ready1 ); +`	`while( ready0 ); +`
`<critical> +`	`<critical> +`
`ready0= FALSE;`	`ready1= FALSE;`
`<noncritical> +`	`<noncritical> +`
`ready0= TRUE;`	`ready1= TRUE;`
`...`	`...`

`+` = mehrfach, mind. einmal
`*` = mehrfach oder gar nicht

◆ Durchspielen aller möglichen Durchmischungen

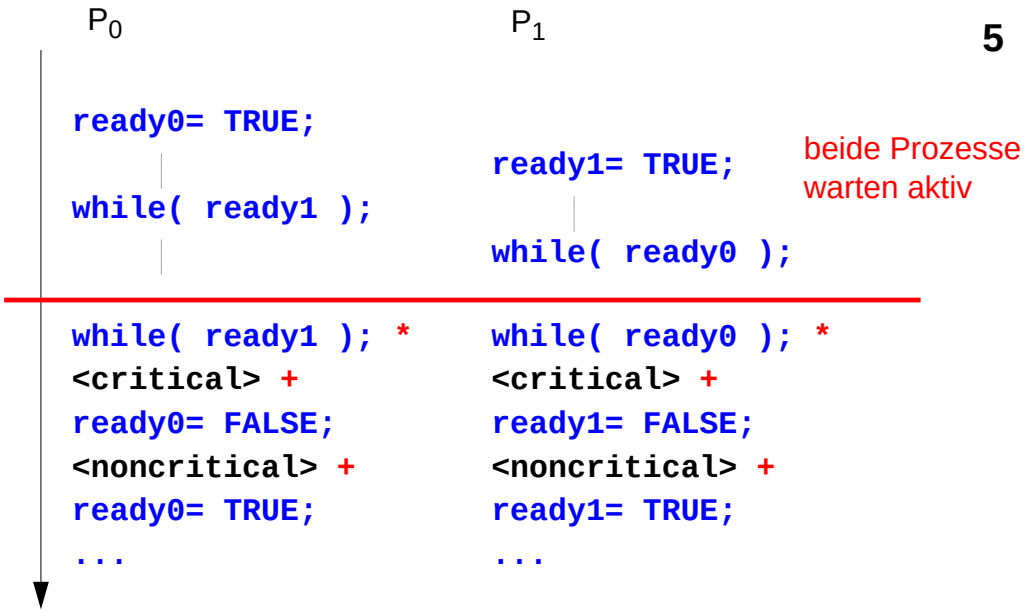
6.1 Gegenseitiger Ausschluss (6)

- Harmlose Durchmischung

P_0	P_1	6
`ready0= TRUE;`		
`while( ready1 );`		
`<critical>`		
	`ready1= TRUE;`	ausgeführte Anweisungen
	`while( ready0 );`	
`<critical> *`	`while( ready0 ); *`	
`ready0= FALSE;`	`<critical> +`	
`<noncritical> +`	`ready1= FALSE;`	
`ready0= TRUE;`	`<noncritical> +`	
`...`	`ready1= TRUE;`	
	`...`	

6.1 Gegenseitiger Ausschluss (7)

- Verklemmung *(Lifelock)*



6.1 Gegenseitiger Ausschluss (8)

- 3. Versuch (Algorithmus von Peterson, 1981)

```
bool ready0= FALSE;
bool ready1= FALSE;
int turn= 0;
```

Prozess 0

```
while( 1 ) {
    ready0= TRUE;
    turn= 1;
    while( ready1 &&
           turn == 1 );

    ... /* critical sec. */

    ready0= FALSE;

    ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
    ready1= TRUE;
    turn= 0;
    while( ready0 &&
           turn == 0 );

    ... /* critical sec. */

    ready1= FALSE;

    ... /* uncritical */
}
```

6.1 Gegenseitiger Ausschluss (9)

- Algorithmus implementiert gegenseitigen Ausschluss
  - vollständige und sichere Implementierung
  - **`turn`** entscheidet für den kritischen Fall von Versuch 2, welcher Prozess nun wirklich den kritischen Abschnitt betreten darf
  - in allen anderen Fällen ist **`turn`** unbedeutend
- Problem der Lösung
  - aktives Warten
- Algorithmus auch für mehrere Prozesse erweiterbar
  - Lösung ist relativ aufwendig

6.2 Spezielle Maschinenbefehle

- Spezielle Maschinenbefehle können die Programmierung kritischer Abschnitte unterstützen und vereinfachen
  - *Test-and-Set* Instruktion
  - *Swap* Instruktion
- Test-and-set
  - Maschinenbefehl mit folgender Wirkung

```
bool test_and_set( bool *plock )
{
    bool tmp= *plock;
    *plock= TRUE;
    return tmp;
}
```

  - Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (2)

◆ Kritische Abschnitte mit Test-and-Set Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

Prozess 1

```
while( 1 ) {
  while(
    test_and_set(&lock) );

  ... /* critical sec. */

  lock= FALSE;

  ... /* uncritical */
}
```

★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.2 Spezielle Maschinenbefehle (3)

■ Swap

◆ Maschinenbefehl mit folgender Wirkung

```
void swap( bool *ptr1, bool *ptr2)
{
    bool tmp= *ptr1;
    *ptr1= *ptr2;
    *ptr2= tmp;
}
```

◆ Ausführung ist atomar

6.2 Spezielle Maschinenbefehle (4)

- Kritische Abschnitte mit Swap-Befehlen

```
bool lock= FALSE;
```

Prozess 0

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

Prozess 1

```
bool key;
...
while( 1 ) {
  key= TRUE;
  while( key == TRUE )
    swap( &lock, &key );

  ... /* critical sec. */

  lock= FALSE;
  ... /* uncritical */
}
```

- ★ Code ist identisch und für mehr als zwei Prozesse geeignet

6.3 Kritik an den bisherigen Verfahren

- ★ Spinlock
  - ◆ bisherige Verfahren werden auch Spinlocks genannt
  - ◆ aktives Warten
- ▲ Problem des aktiven Wartens
  - ◆ Verbrauch von Rechenzeit ohne Nutzen
  - ◆ Behinderung „nützlicher“ Prozesse
  - ◆ Abhängigkeit von der Schedulingstrategie
    - nicht anwendbar bei nicht-verdrängenden Strategien
    - schlechte Effizienz bei langen Zeitscheiben
- Spinlocks kommen heute fast ausschließlich in Multiprozessorsystemen zum Einsatz
  - ◆ bei kurzen kritischen Abschnitten effizient
  - ◆ Koordinierung zwischen Prozessen von mehreren Prozessoren

6.4 Sperrung von Unterbrechungen

- Sperrung der Systemunterbrechungen im Betriebssystems

Prozess 0

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

Prozess 1

```
disable_interrupts();

... /* critical sec. */

enable_interrupts();

... /* uncritical sec. */
```

- ◆ nur für kurze Abschnitte geeignet
  - sonst Datenverluste möglich
- ◆ nur innerhalb des Betriebssystems möglich
  - privilegierter Modus nötig
- ◆ nur für Monoprozessoren anwendbar
  - bei Multiprozessoren arbeiten andere Prozesse echt parallel

6.5 Semaphor

- Ein Semaphor (griech. Zeichenträger) ist eine Datenstruktur des Systems mit zwei Operationen (nach *Dijkstra*)
  - ◆ P-Operation *(proberen; passeren; wait; down)*
    - wartet bis Zugang frei

```
void P( int *s )
{
    while( *s <= 0 );
    *s= *s-1;
}
```

atomare Funktion

  - ◆ V-Operation *(verhogen; vrijgeven; signal; up)*
    - macht Zugang für anderen Prozess frei

```
void V( int *s )
{
    *s= *s+1;
}
```

atomare Funktion

6.5 Semaphor (2)

- Implementierung kritischer Abschnitte mit einem Semaphor

```
int lock= 1;
```

Prozess 0

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

Prozess 1

```
...
while( 1 ) {
  P( &lock );

  ... /* critical sec. */

  V( &lock );

  ... /* uncritical */
}
```

- Problem:
  - Implementierung von P und V

6.5 Semaphor (3)

- Implementierung im Betriebssystem (Monoprozessor)

P-Operation

Unterbrechungen sperren		
	***s <= 0**	
	ja	nein
ok = FALSE;		
laufenden Prozess in Zustand *blockiert* versetzen und in Warteschlange aufnehmen		***s = *s-1** **ok = TRUE;**
Prozessumschalter ansteuern und Unterbrechungen freigeben		Unterbrechungen freigeben
until: ok == TRUE		

ok ist eine prozesslokale Variable

- jeder Semaphor besitzt Warteschlange, die blockierte Prozesse aufnimmt

6.5 Semaphor (4)

V-Operation

Unterbrechungen sperren
`*s = *s+1`
alle Prozess aus der Warteschlange in den Zustand *bereit* versetzen
Unterbrechungen freigeben
Prozessumschalter ansteuern

- Prozesse probieren immer wieder, die P-Operation erfolgreich abzuschließen
- Schedulingstrategie entscheidet über Reihenfolge und Fairness
  - leichte Ineffizienz durch Aufwecken aller Prozesse
  - mit Einbezug der Schedulingstrategie effizientere Implementierungen möglich

6.5 Semaphor (5)

★ Vorteile einer Semaphor-Implementierung im Betriebssystem
- Einbeziehen des Schedulers in die Semaphor-Operationen
- kein aktives Warten; Ausnutzen der Blockierzeit durch andere Prozesse

■ Implementierung einer Synchronisierung
- zwei Prozesse P_1 und P_2
- Anweisung S_1 in P_1 soll vor Anweisung S_2 in P_2 stattfinden

```
int lock= 0;
```

Prozess 1
```
...
S1;
V( &lock );
...
```

Prozess 2
```
...
P( &lock );
S2;
...
```

★ Zählende Semaphore

6.5 Semaphor (6)

- Abstrakte Beschreibung von zählenden Semaphoren (PV System)
  - ◆ für jede Operation wird eine Bedingung angegeben
    - falls Bedingung nicht erfüllt, wird die Operation blockiert
  - ◆ für den Fall, dass die Bedingung erfüllt wird, wird eine Anweisung definiert, die ausgeführt wird
- Beispiel: zählende Semaphore

Operation	Bedingung	Anweisung
P(S)	S > 0	S := S – 1
V(S)	TRUE	S := S + 1

7 Klassische Koordinierungsprobleme

- Reihe von bedeutenden Koordinierungsproblemen
  - ◆ Gegenseitiger Ausschluss *(Mutual exclusion)*
    - nur ein Prozess darf bestimmte Anweisungen ausführen
  - ◆ Puffer fester Größe *(Bounded buffers)*
    - Blockieren der lesenden und schreibenden Prozesse, falls Puffer leer oder voll
  - ◆ Leser-Schreiber-Problem *(Reader-writer problem)*
    - Leser können nebenläufig arbeiten; Schreiber darf nur alleine zugreifen
  - ◆ Philosophenproblem *(Dining-philosopher problem)*
    - im Kreis sitzende Philosophen benötigen das Besteck der Nachbarn zum Essen
  - ◆ Schlafende Friseure *(Sleeping-barber problem)*
    - Friseure schlafen solange keine Kunden da sind

7.1 Gegenseitiger Ausschluss

- Semaphor
  - eigentlich reicht ein Semaphor mit zwei Zuständen: binärer Semaphor

```
void P( int *s )
{
    while( *s == 0 );
    *s= 0;
}
```

atomare Funktion

```
void V( int *s )
{
    *s= 1;
}
```

atomare Funktion

  - zum Teil effizienter implementierbar

7.1 Gegenseitiger Ausschluss (2)

- Abstrakte Beschreibung: binäre Semaphore

Operation	Bedingung	Anweisung
P(S)	$S \neq 0$	S := 0
V(S)	TRUE	S := 1

7.1 Gegenseitiger Ausschluss (3)

- Problem der Klammerung kritischer Abschnitte
  - Programmierer müssen Konvention der Klammerung einhalten
  - Fehler bei Klammerung sind fatal

```
P( &lock );
... /* critical sec. */
P( &lock );
```

führt zu Verklemmung (Deadlock)

```
V( &lock );
... /* critical sec. */
V( &lock );
```

führt zu unerwünschter Nebenläufigkeit

7.1 Gegenseitiger Ausschluss (3)

- Automatische Klammerung wünschenswert
  - Beispiel: Java

```
synchronized( lock ) {
... /* critical sec. */
}
```

7.2 Bounded Buffers

- Puffer fester Größe
  - mehrere Prozesse lesen und beschreiben den Puffer
  - beispielsweise Erzeuger und Verbraucher (Erzeuger-Verbraucher-Problem)
    (z.B. Erzeuger liest einen Katalog; Verbraucher zählt Zeilen; Gesamtanwendung zählt Einträge in einem Katalog)
  - UNIX-Pipe ist solch ein Puffer
- Problem
  - Koordinierung von Leser und Schreiber
    - gegenseitiger Ausschluss beim Pufferzugriff
    - Blockierung des Lesers bei leerem Puffer
    - Blockierung des Schreibers bei vollem Puffer

7.2 Bounded Buffers (2)

- Implementierung mit zählenden Semaphoren
  - zwei Funktionen zum Zugriff auf den Puffer
    - **`put`** stellt Zeichen in den Puffer
    - **`get`** liest ein Zeichen vom Puffer
  - Puffer wird durch ein Feld implementiert, das als Ringpuffer wirkt
    - zwei Integer-Variablen enthalten Feldindizes auf den Anfang und das Ende des Ringpuffers
  - ein Semaphor für den gegenseitigen Ausschluss
  - je einen Semaphor für das Blockieren an den Bedingungen „Puffer voll“ und „Puffer leer“
    - Semaphor **`full`** zählt wieviele Zeichen noch in den Puffer passen
    - Semaphor **`empty`** zählt wieviele Zeichen im Puffer sind

7.2 Bounded Buffers (3)

```
char buffer[N];
int inslot= 0, outslot= 0;
semaphor mutex= 1, empty= 0, full= N;
```

```
void put( char c )
{
    P( &full );
    P( &mutex );
    buffer[inslot]= c;
    if( ++inslot >= N )
        inslot= 0;
    V( &mutex );
    V( &empty );
}
```

```
char get( void )
{
    char c;

    P( &empty );
    P( &mutex );
    c= buffer[outslot];
    if( ++outslot >= N )
        outslot= 0;
    V( &mutex );
    V( &full );
    return c;
}
```


7.3 Erstes Leser-Schreiber-Problem

- Lesende und schreibende Prozesse
  - Leser können nebenläufig zugreifen (Leser ändern keine Daten)
  - Schreiber können nur exklusiv zugreifen (Daten sonst inkonsistent)
- Erstes Leser-Schreiber-Problem (nach *Courtois* et.al. 1971)
  - Kein Leser soll warten müssen, es sei denn ein Schreiber ist gerade aktiv
- Realisierung mit zählenden (binären) Semaphoren
  - Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreiber gegen Leser: **`write`**
  - Zählen der nebenläufig tätigen Leser: Variable **`readcount`**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **`readcount`**: **`mutex`**

7.3 Erstes Leser-Schreiber-Problem (2)

```
semaphore mutex= 1, write= 1;
int readcount= 0;
```

Leser

```
...
P( &mutex );
if( ++readcount == 1 )
    P( &write );
V( &mutex );

... /* reading */

P( &mutex );
if( --readcount == 0 )
    V( &write );
V( &mutex );
...
```

Schreiber

```
...
P( &write );

... /* writing */

V( &write );
...
```


7.3 Erstes Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
  - PV-Chunk Semaphore:
    führen quasi mehrere P- oder V-Operationen atomar aus
    - zweiter Parameter gibt Anzahl an
- Abstrakte Beschreibung für PV-Chunk Semaphore:

Operation	Bedingung	Anweisung
P(S, k)	$S \geq k$	S := S – k
V(S, k)	TRUE	S := S + k

7.3 Erstes Leser-Schreiber-Problem (4)

- Implementierung mit PV-Chunk:
  - Annahme: es gibt maximal **N** Leser

```
PV_chunk_semaphore mutex= N;
```

Leser

```
...
Pc( &mutex, 1 );

... /* reading */

Vc( &mutex, 1 );
...
```

Schreiber

```
...
Pc( &mutex, N );

... /* writing */

Vc( &mutex, N );
...
```


7.4 Zweites Leser-Schreiber-Problem

- Wie das erste Problem aber: (nach Courtois et.al., 1971)
  - Schreiboperationen sollen so schnell wie möglich durchgeführt werden
- Implementierung mit zählenden Semaphoren
  - Zählen der nebenläufig tätigen Leser: Variable **readcount**
  - Zählen der anstehenden Schreiber: Variable **writecount**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **readcount**: **mutexR**
  - Semaphor für gegenseitigen Ausschluss beim Zugriff auf **writecount**: **mutexW**
  - Semaphor für gegenseitigen Ausschluss von Schreibern untereinander und von Schreibern gegen Leser: **write**
  - Semaphor für den Ausschluss von Lesern, falls Schreiber vorhanden: **read**
  - Semaphor zum Klammern des Leservorspanns: **mutex**

7.4 Zweites Leser-Schreiber-Problem (2)

```
semaphore mutexR= 1, mutexW= 1, mutex= 1;
semaphore write= 1, read= 1;
int readcount= 0, writecount= 0;
```

Bitte nicht versuchen, dies zu verstehen!!

Leser

```
...
P( &mutex ); P( &read );
  P( &mutexR );
  if( ++readcount == 1 )
    P( &write );
  V( &mutexR );
V( &read ); V( &mutex );

... /* reading */

P( &mutexR );
if( --readcount == 0 )
    V( &write );
V( &mutexR );
...
```

Schreiber

```
...
P( &mutexW );
if( ++writecount == 1 )
  P( &read );
V( &mutexW );
P( &write );

... /* writing */

V( &write );
P( &mutexW );
if( --writecount == 0 )
    V( &read );
V( &mutexW );
...
```


7.4 Zweites Leser-Schreiber-Problem (3)

- Vereinfachung der Implementierung durch spezielle Semaphore?
  - Up-Down–Semaphore:
    - zwei Operationen *up* und *down,* die den Semaphor hoch- und runterzählen
    - Nichtblockierungsbedingung für beide Operationen, definiert auf einer Menge von Semaphoren

7.4 Zweites Leser-Schreiber-Problem (4)

- Abstrakte Beschreibung für Up-down–Semaphore

Operation	Bedingung	Anweisung
up(S, { S_i })	$\sum_i S_i \geq 0$	S := S + 1
down(S, { S_i })	$\sum_i S_i \geq 0$	S := S – 1

7.4 Zweites Leser-Schreiber-Problem (5)

- Implementierung mit Up-Down–Semaphoren:

```
up_down_semaphor mutexw= 0, reader= 0, writer= 0;
```

Leser

```
...
down( &reader, 1, &writer );

... /* reading */

up( &reader, 0 );
...
```

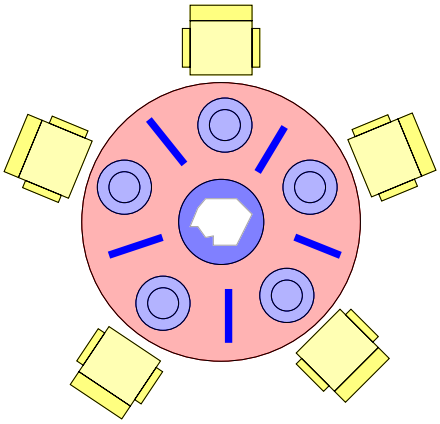
Schreiber

```
...
down( &writer, 0 );
down( &mutexw,
  2, &mutexw,&reader );

... /* writing */

up( &mutexw, 0 );
up( &writer, 0 );
...
```

- Zähler für Leser: **`reader`** (zählt negativ)
- Zähler für anstehende Schreiber: **`writer`** (zählt negativ)
- Semaphor für gegenseitigen Ausschluss der Schreiber: **`mutexw`**

7.5 Philosophenproblem

- Fünf Philosophen am runden Tisch



  - Philosophen denken oder essen
    "The life of a philosopher consists of an alternation of thinking and eating." (Dijkstra, 1971)
  - zum Essen benötigen sie zwei Gabeln, die jeweils zwischen zwei benachbarten Philosophen abgelegt sind

- Problem
  - Gleichzeitiges Belegen mehrerer Betriebsmittel (hier Gabeln)
  - Verklemmung und Aushungerung

7.5 Philosophenproblem (2)

- Naive Implementierung
  - eine Semaphor pro Gabel

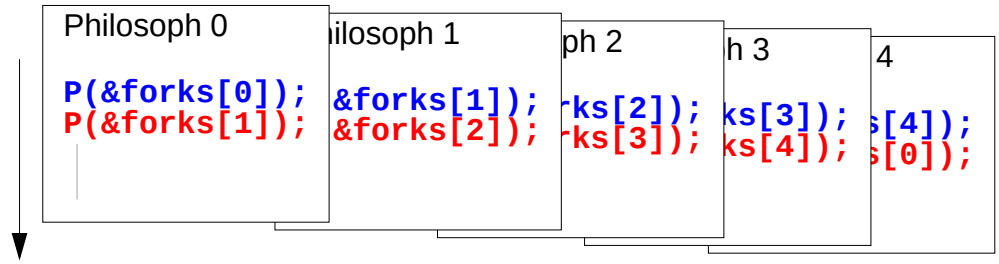
```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, i ∈ [0,4]

```
while( 1 ) {
    ... /* think */

    P( &forks[i] );
    P( &forks[(i+1)%5] );

    ... /* eat */

    V( &forks[i] );
    V( &forks[(i+1)%5] );
}
```

7.5 Philosophenproblem (3)

- Problem der Verklemmung
  - alle Philosophen nehmen gleichzeitig die linke Gabel auf und versuchen dann die rechte Gabel aufzunehmen



  - System ist verklemmt
    - Philosophen warten alle auf ihre Nachbarn

7.5 Philosophenproblem (4)

- Lösung 1: gleichzeitiges Aufnehmen der Gabeln
  - Implementierung mit binären oder zählenden Semaphoren ist nicht trivial
  - Zusatzvariablen erforderlich
  - unübersichtliche Lösung
- ★ Einsatz von speziellen Semaphoren: PV-multiple–Semaphore
  - gleichzeitiges und atomares Belegen mehrerer Semaphoren
  - Abstrakte Beschreibung:

Operation	Bedingung	Anweisung
P({ S_i })	$\forall i, S_i > 0$	$\forall i, S_i = S_i - 1$
V({ S_i })	TRUE	$\forall i, S_i = S_i + 1$

7.5 Philosophenproblem (5)

◆ Implementierung mit PV-multiple–Semaphoren

```
PV_mult_semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, i ∈ [0,4]

```
while( 1 ) {
    ... /* think */

    Pm( 2, &forks[i], &forks[(i+1)%5] );

    ... /* eat */

    Vm( 2, &forks[i], &forks[(i+1)%5] );
}
```


7.5 Philosophenproblem (6)

■ Lösung 2: einer der Philosophen muss erst die andere Gabel aufnehmen

```
semaphor forks[5]= { 1, 1, 1, 1, 1 };
```

Philosoph i, i ∈ [0,3]

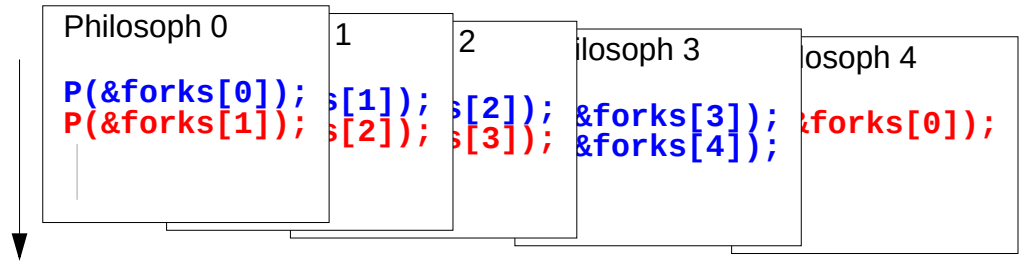
```
while( 1 ) {
    ... /* think */

    P( &forks[i] );
    P( &forks[(i+1)%5] );

    ... /* eat */

    V( &forks[i] );
    V( &forks[(i+1)%5] );
}
```

Philosoph 4

```
while( 1 ) {
    ... /* think */

    P( &forks[0] );
    P( &forks[4] );

    ... /* eat */

    V( &forks[0] );
    V( &forks[4] );
}
```

7.5 Philosophenproblem (7)

- Ablauf der asymmetrischen Lösung im ungünstigsten Fall



- System verklemmt sich nicht

7.6 Schlafende Friseure

- Friseurladen mit N freien Wartestühlen
  - Friseure schlafen solange kein Kunde da ist
  - eintretende Kunden warten bis ein Friseur frei ist; gegebenenfalls wird einer der Friseure von einem Kunden aufgeweckt
  - sind keine Wartestühle mehr frei, verlassen die Kunden den Laden
- Problem:
  - Mehrere Bearbeitungsstationen sollen exklusive Bearbeitungen durchführen
- Implementierung mit zählenden Semaphoren
  - Semaphore zum Schutz der Variablen zum Zählen der Kunden: **`mutex`**
  - Semaphore zum Zählen der Friseure: **`barbers`**
  - Semaphore zum Zählen der Kunden: **`customers`**

7.6 Schlafende Friseure (2)

- Implementierung mit zählenden Semaphoren (PV System)

```
semaphor customers= 0, barbers= 0, mutex= 1;
int waiting= 0;
```

Barber

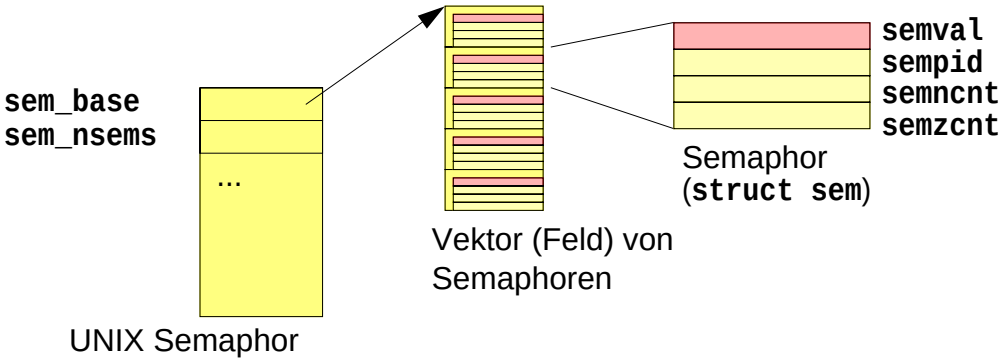
```
while( 1 ) {
    P( &customers );
    P( &mutex );
    waiting--;
    V( &barbers );
    V( &mutex );

    ... /* cut hair */
}
```

Customer

```
P( &mutex );
if( waiting < N ) {
    waiting++
    V( &customers );
    V( &mutex );
    P( &barbers );

    ... /* get hair cut */
}
else {
    V( &mutex );
}
```


8 UNIX-Semaphor

- Ein UNIX-Semaphor entspricht einem Vektor von Einzelsemaphoren (erweitertes Vektoradditionssystem)

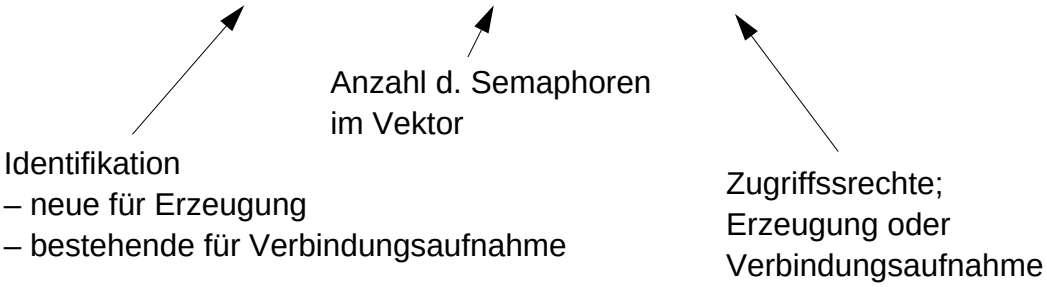


- Gleichzeitige und atomare Operationen auf mehreren Semaphoren im Vektor möglich

8.1 Erzeugen einer UNIX-Semaphore

- UNIX-Semphore haben systemweit eindeutige Identifikation (Key)
  - Erzeugen und Aufnehmen der Verbindung zu einer Semaphore

```
int semget( key_t key, int nsems, int semflg );
```



  - Ergebnis ist eine Semaphore ID ähnlich wie ein Filedescriptor
    - Semaphore ID muss bei allen Operationen verwendet werden
  - Zugriffsrechte: Lesen, Verändern
    - einstellbar für Besitzer, Gruppe und alle anderen (ähnlich wie bei Dateien)

8.1 Erzeugen einer UNIX-Semaphore (2)

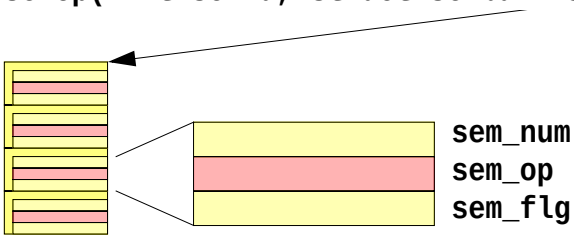
- Verwendung des Keys
  - Alle Prozesse, die auf die Semaphore zugreifen wollen, müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann keine Semaphore mit gleichem Key erzeugt werden
  - Ist ein Key bekannt, kann auf die Semaphore zugegriffen werden
    - gesetzte Zugriffsberechtigungen werden allerdings beachtet
  - Private Semaphoren (ohne Key) können erzeugt werden
- Semaphore sind persistent
  - Explizites Löschen notwendig

```
ipcrm -S <key>
```

8.2 Operationen auf UNIX-Semphoren

- Operationen auf mehreren der Semaphoren im Vektor

```
int semop( int semid, struct sembuf *sops, size_t nsops );
```



Anzahl der Einzel-operationen

◆ Operationen

- **sem_num**: Nummer des Semaphor im Vektor
- **sem_op < 0**: ähnlich P-Operation – Herunterzählen des Semaphor (blockierend oder mit Fehlerstatus, je nach **sem_flg**)
- **sem_op > 0**: ähnlich V-Operation – Hochzählen des Semaphore
- **sem_op == 0**: Test auf 0 (blockierend oder mit Fehlerstatus, je nach **sem_flg**)

8.2 Operationen auf UNIX-Semaphoren (2)

- Kontrolloperationen

```
int semctl( int semid, int semnum, int cmd,
            [ union semun arg ] );
```

◆ explizites Setzen von Werten (einen, alle)

◆ Abfragen von Werten (einen, alle)

◆ Abfragen von Zusatzinformationen

- welcher Prozess hat letzte Operation erfolgreich durchgeführt
- wann wurde letzte Operation durchgeführt
- Zugriffsrechte
- Anzahl der blockierten Prozesse

◆ Löschen des Semaphor

8.3 Beispiel: Philosophenproblem

- Ein UNIX-Semaphor mit fünf Elementen (entsprechen Gabeln)
  - Deklarationen

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int i;                          /* number of philosopher */
int j;
int semid;                          /* semaphor ID */
struct sembuf pbuf[2], vbuf[2]; /* operation buffer */

union semun {                   /* UNION for semctl */
    int val;
    struct semid_ds *buf;
    ushort *array;
} arg;

...
```


8.3 Beispiel: Philosophenproblem (2)

- Erzeuge Semaphor

```
...

semid= semget( IPC_PRIVATE, 5, IPC_CREAT|SEM_A|SEM_R );
if( semid < 0 ) { ... /* error */ }

for( j= 0; j < 5; j++ ) {   /* set all values to 1 */
    arg.val= 1;
    if( semctl( semid, j, SETVAL, arg ) < 0 ) {
        ... /* error */
    }
}

...
```

8.3 Beispiel: Philosophenproblem (3)

◆ Erzeugen der Prozesse

```
...

for( i=0; i<=3; i++ ) {     /* start children i= 0..3; */
    pid_t pid= fork();

    if( pid < (pid_t)0 ) { ... /* error */ }
    if( pid ==(pid_t)0 ) {
        /* child */

        break;
    }
}                           /* parent: i= 4; */

...
```


8.3 Beispiel: Philosophenproblem (4)

◆ Initialisierungen

```
...     /* we are philosopher i */

/* initialize buffer for P operation */

pbuf[0].sem_num= i; pbuf[1].sem_num= (i+1)%5;
pbuf[0].sem_op= pbuf[1].sem_op= -1;
pbuf[0].sem_flg= pbuf[1].sem_flg= 0;

/* initialize buffer for V operation */

vbuf[0].sem_num= i; vbuf[1].sem_num= (i+1)%5;
vbuf[0].sem_op= vbuf[1].sem_op= 1;
vbuf[0].sem_flg= vbuf[1].sem_flg= 0;

...
```

8.3 Beispiel: Philosophenproblem (5)

- Philosoph

```
...

while( 1 ) {
    ...     /* thinking */

    if( semop( semid, pbuf, 2 ) < 0 ) { ... /* error */ }

    ...     /* eating */

    if( semop( semid, vbuf, 2 ) < 0 ) { ... /* error */ }
}
```


9 Zusammenfassung

- Programmiermodell: Prozess
  - Zerlegung von Anwendungen in Prozesse oder Threads
  - Ausnutzen von Wartezeiten; Time sharing–Betrieb
  - Prozess hat verschiedene Zustände: laufend, bereit, blockiert etc.
- Auswahlstrategien für Prozesse
  - FCFS, SJF, PSJF, RR, MLFB
- Prozesskommunikation
  - Pipes, Queues, Signals, Sockets, Shared memory, RPC
- Koordinierung von Prozessen
  - Einschränkung der gleichzeitigen Abarbeitung von Befehlsfolgen in nebenläufigen Prozessen/Aktivitätsträgern

9 Zusammenfassung (2)

- Gegenseitiger Ausschluss mit Spinlocks
- Klassische Koordinierungsprobleme und deren Lösung mit Semaphoren
  - Gegenseitiger Ausschluss
  - Bounded buffers
  - Leser-Schreiber-Probleme
  - Philosophenproblem
  - Schlafende Friseure

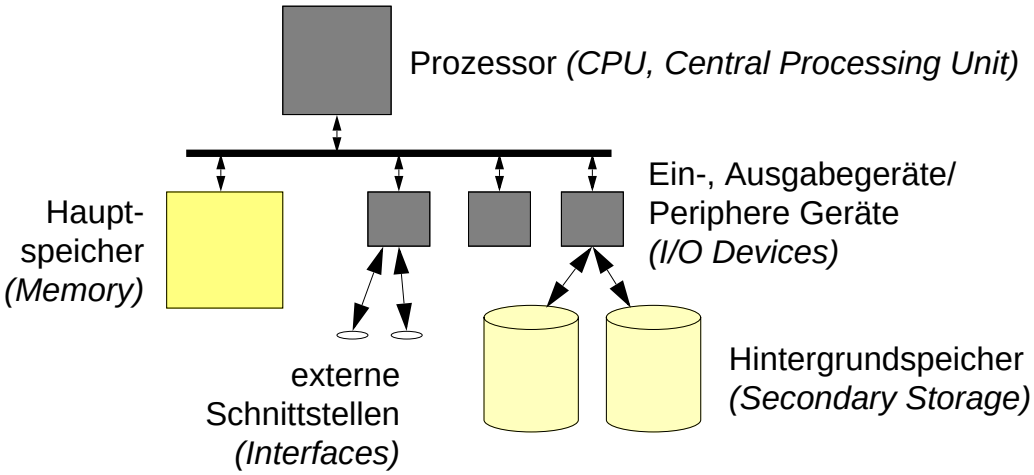
9 Zusammenfassung (3)

- UNIX Systemaufrufe
  - fork, exec, wait, nice
  - pipe, socket, bind, recvfrom, sendto, listen, accept
  - msgget, msgsnd, msgrcv
  - signal, kill, sigaction
  - semget, semop, semctl

E Speicherverwaltung

E Speicherverwaltung

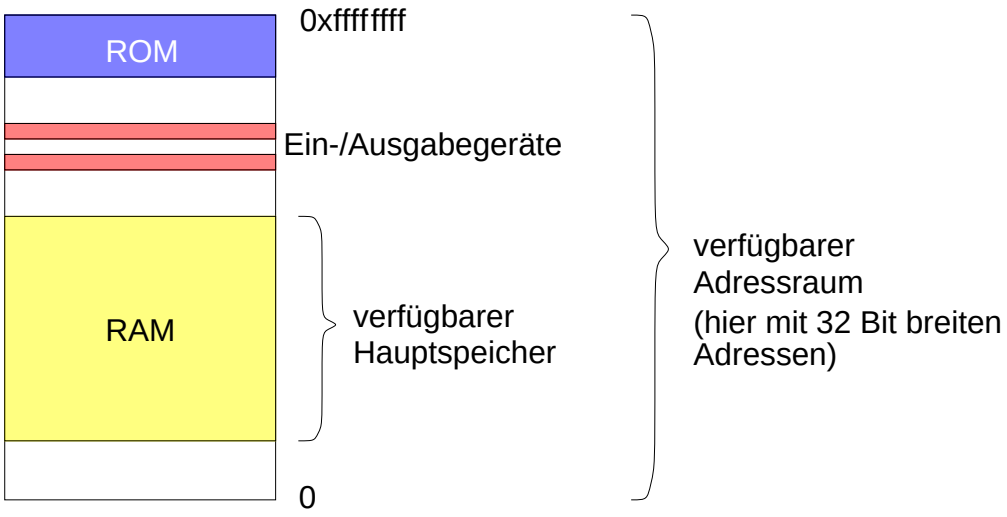
- Betriebsmittel



1 Speichervergabe

1.1 Problemstellung

- Verfügbarer Speicher



1.1 Problemstellung (2)

- Belegung des verfügbaren Hauptspeichers durch
  - Benutzerprogramme
    - Programmbefehle (Code, Binary)
    - Programmdaten
  - Betriebssystem
    - Betriebssystemcode
    - Puffer
    - Systemvariablen
- ★ Zuteilung des Speichers nötig

1.2 Statische Speicherzuteilung

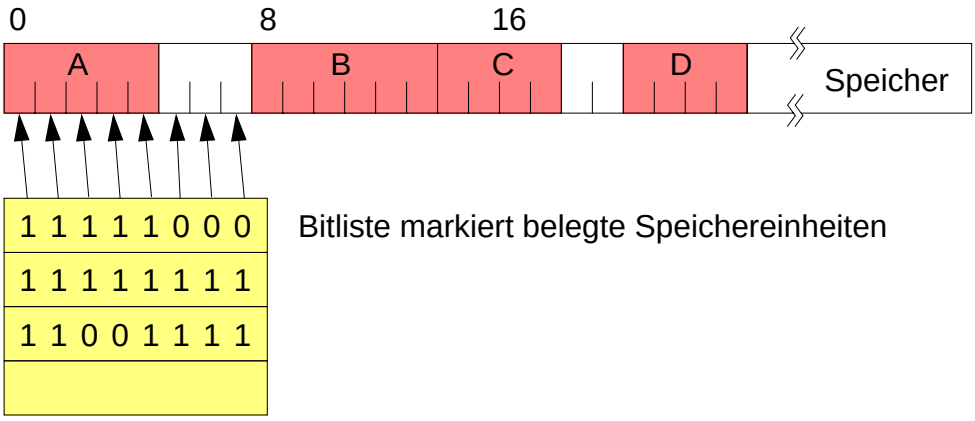
- Feste Bereiche für Betriebssystem und Benutzerprogramm
- Probleme:
  - Begrenzung anderer Ressourcen
    (z.B. Bandbreite bei Ein-/Ausgabe wg. zu kleiner Systempuffer)
  - Ungenutzter Speicher des Betriebssystems kann von Anwendungsprogramm nicht genutzt werden und umgekehrt
- Dynamische Speicherzuteilung einsetzen

1.3 Dynamische Speicherzuteilung

- Segmente
  - zusammenhängender Speicherbereich
    (Bereich mit aufeinanderfolgenden Adressen)
- Allokation (Anforderung) und Freigabe von Segmenten
- Ein Anwendungsprogramm besitzt üblicherweise folgende Segmente (siehe auch D.2.4):
  - Codesegment
  - Datensegment
  - Stacksegment (für Verwaltungsinformationen, z.B. bei Funktionsaufrufen)
- Suche nach geeigneten Speicherbereichen zur Zuteilung
- Speicherzuteilungsstrategien nötig

1.4 Freispeicherverwaltung

- Freie (evtl. auch belegte) Segmente des Speichers müssen repräsentiert werden
- Bitlisten

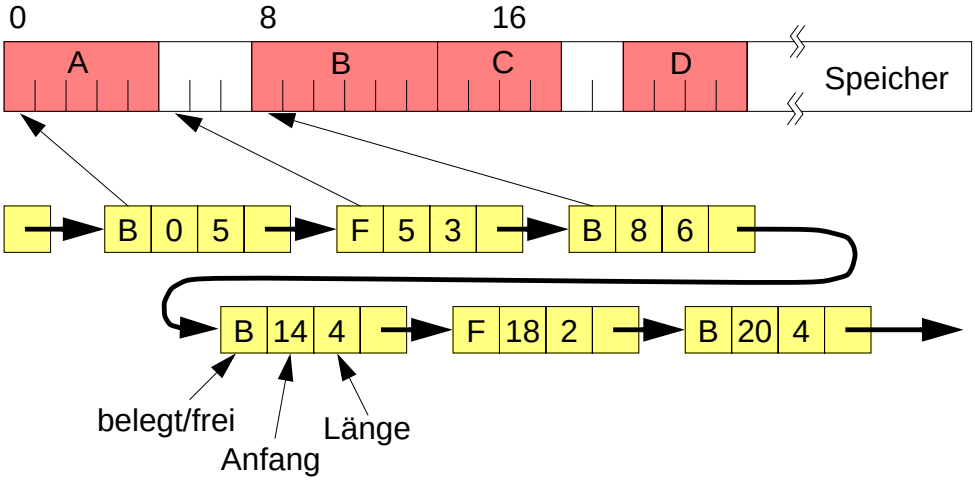


Bitliste markiert belegte Speichereinheiten

Speichereinheiten gleicher Größe (z.B. 1 Byte, 64 Byte, 1024 Byte)

1.4 Freispeicherverwaltung (2)

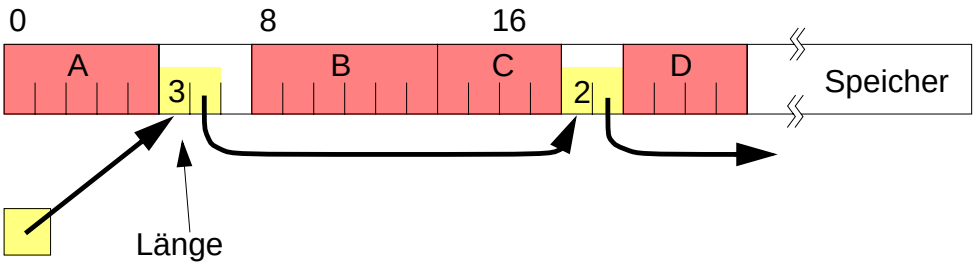
- Verkettete Liste



Repräsentation auch von freien Segmenten

1.4 Freispeicherverwaltung (3)

- Verkettete Liste in dem freien Speicher

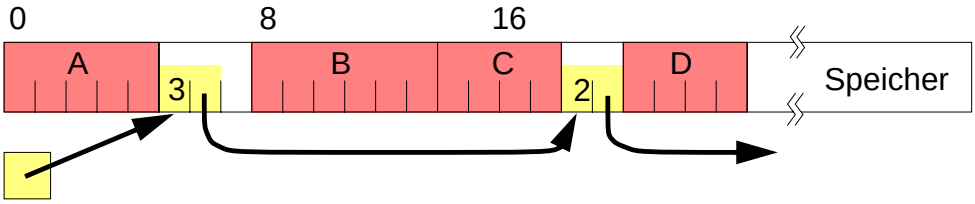


Mindestlückengröße muss garantiert werden

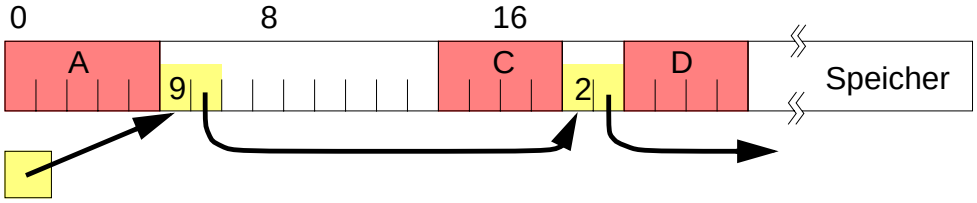
- Zur Effizienzsteigerung eventuell Rückwärtsverkettung nötig
- Repräsentation letztlich auch von der Vergabestrategie abhängig

1.5 Speicherfreigabe

- Verschmelzung von Lücken



nach Freigabe von B:

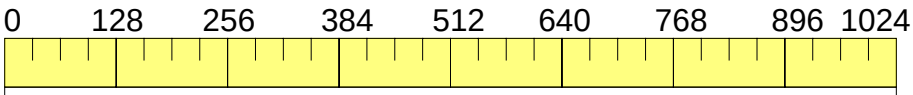


1.6 Vergabestrategien

- First Fit
  - ◆ erste passende Lücke wird verwendet
- Rotating First Fit / Next Fit
  - ◆ wie First Fit aber Start bei der zuletzt zugewiesenen Lücke
- Best Fit
  - ◆ kleinste passende Lücke wird gesucht
- Worst Fit
  - ◆ größte passende Lücke wird gesucht
- ▲ Probleme:
  - ◆ Speicherverschnitt
  - ◆ zu kleine Lücken

1.7 Buddy Systeme

- Einteilung in dynamische Bereiche der Größe 2^n



	1024					
Anfrage 70	A	128		256		512
Anfrage 35	A	B	64	256		512
Anfrage 80	A	B	64	C	128	512
Freigabe A	128	B	64	C	128	512
Anfrage 60	128	B	D	C	128	512
Freigabe B	128	64	D	C	128	512
Freigabe D	256			C	128	512
Freigabe C	1024					

Effiziente Repräsentation der Lücken und effiziente Algorithmen

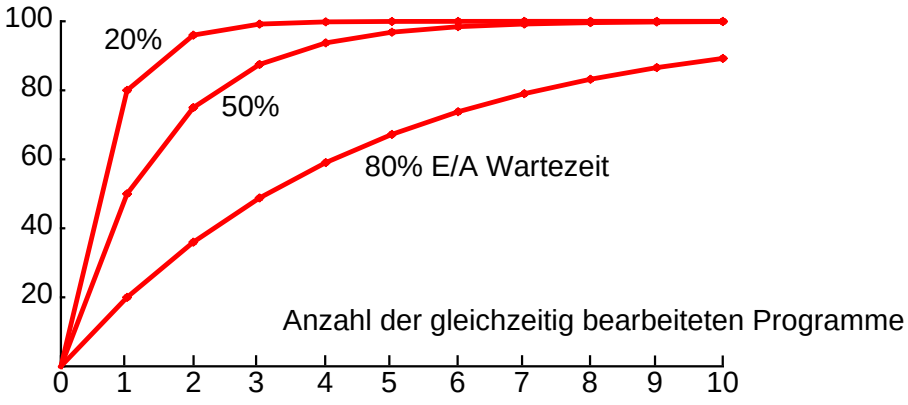
1.8 Einsatz der Verfahren

- Einsatz im Betriebssystem
  - Verwaltung des Systemspeichers
  - Zuteilung von Speicher an Prozesse und Betriebssystem
- Einsatz innerhalb eines Prozesses
  - Verwaltung des Haldenspeichers *(Heap)*
  - erlaubt dynamische Allokation von Speicherbereichen durch den Prozess (**`malloc`** und **`free`**)
- Einsatz für Bereiche des Sekundärspeichers
  - Verwaltung bestimmter Abschnitte des Sekundärspeichers z.B. Speicherbereich für Prozessauslagerungen *(Swap space)*

2 Mehrprogrammbetrieb

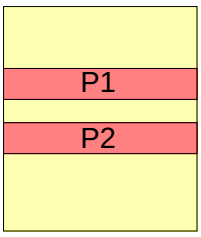
2.1 Problemstellung

- Mehrere Prozesse laufen gleichzeitig
  - Wartezeiten von Ein-/Ausgabeoperationen ausnutzen
  - CPU Auslastung verbessern
  - CPU-Nutzung in Prozent, abhängig von der Anzahl der Prozesse



2.1 Problemstellung (2)

- Mehrere Prozesse benötigen Hauptspeicher
  - Prozesse liegen an verschiedenen Stellen im Hauptspeicher.
  - Speicher reicht eventuell nicht für alle Prozesse.
  - Schutzbedürfnis des Betriebssystems und der Prozesse untereinander



zwei Prozesse und deren Codesegmente im Speicher

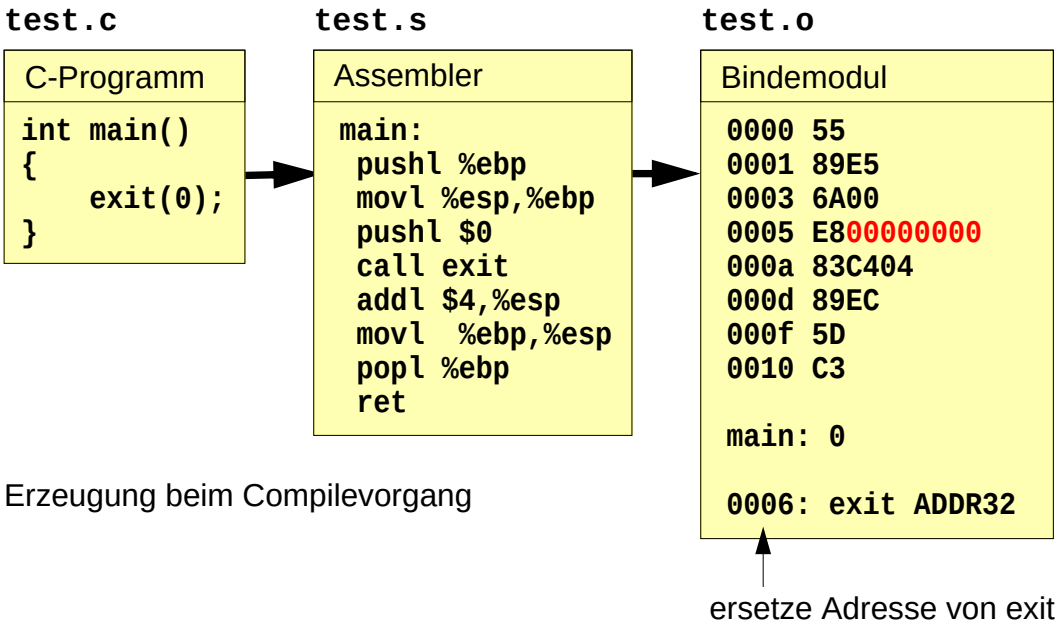
- Relokation von Programmbefehlen (Binaries)
- Ein- und Auslagern von Prozessen
- Hardwareunterstützung

2.2 Relokation

- Festlegung absoluter Speicheradressen in den Programmbefehlen
  - z.B. ein Sprungbefehl in ein Unterprogramm oder
    ein Ladebefehl für eine Variable aus dem Datensegment
- Absolutes Binden (*Compile Time*)
  - Adressen stehen fest
  - Programm kann nur an bestimmter Speicherstelle korrekt ablaufen
- Statisches Binden (*Load Time*)
  - Beim Laden (Starten) des Programms werden die absoluten Adressen angepasst (reloziert)
  - Relokationsinformation nötig, die vom Compiler oder Assembler geliefert wird

2.2 Relokation (2)

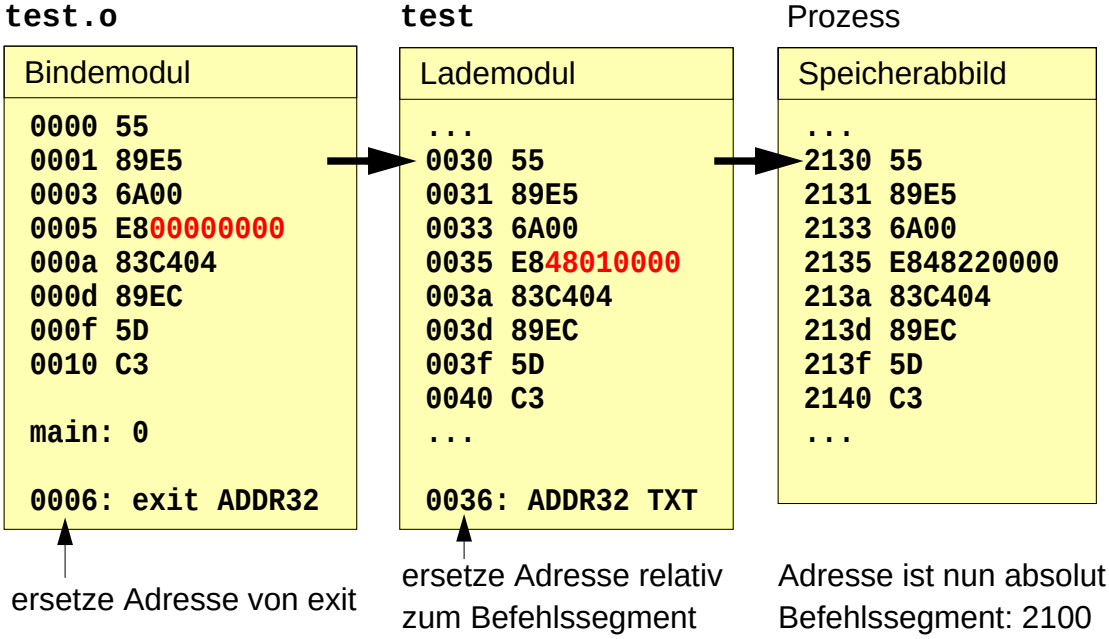
- Compilevorgang (Erzeugung der Relokationsinformation)



Erzeugung beim Compilevorgang

2.2 Relokation (3)

- Binde- und Ladevorgang

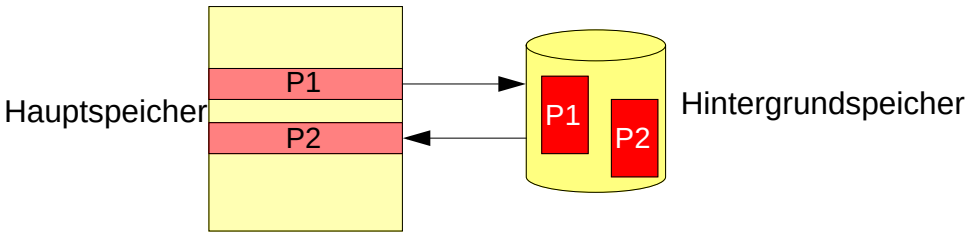


2.2 Relokation (4)

- Relokationsinformation im Bindemodul
  - erlaubt das Binden von Modulen in beliebige Programme
- Relokationsinformation im Lademodul
  - erlaubt das Laden des Programms an beliebige Speicherstellen
  - absolute Adressen werden erst beim Laden generiert
- ★ Alternative
  - Programm benutzt keine absoluten Adressen und kann daher immer an beliebige Speicherstellen geladen werden

2.3 Ein-, Auslagerung (*Swapping*)

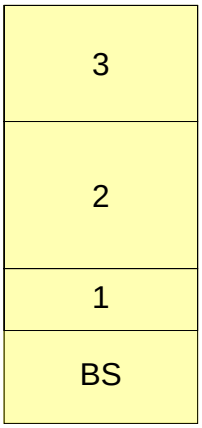
- Segmente eines Prozesses werden auf Hintergrundspeicher ausgelagert und im Hauptspeicher freigegeben
  - z.B. zur Überbrückung von Wartezeiten bei E/A oder Round-Robin Schedulingstrategie
- Einlagern der Segmente in den Hauptspeicher am Ende der Wartezeit



- ▲ Aus-, Einlagerzeit ist hoch
  - Latenzzeit der Festplatte
  - Übertragunszeit

2.3 Ein-, Auslagerung (2)

- Prozess ist statisch gebunden
  - kann nur an gleiche Stelle im Hauptspeicher wieder eingelagert werden
  - Kollisionen mit eventuell neu im Hauptspeicher befindlichen Segmenten
- Mögliche Lösung: Partitionierung des Hauptspeichers

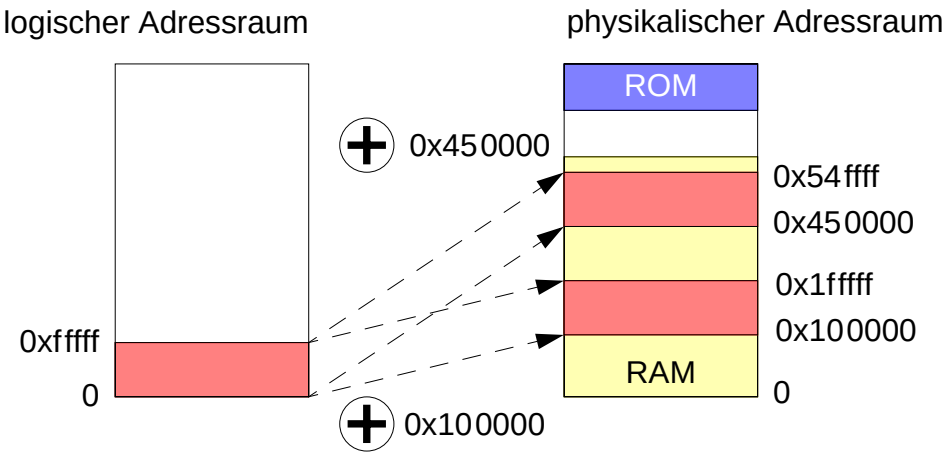


Partition 3
Partition 2
Partition 1
Betriebssystem

  - In jeder Partition läuft nur ein Prozess
  - Einlagerung erfolgt wieder in die gleiche Partition
  - Speicher kann nicht optimal genutzt werden

2.4 Segmentierung

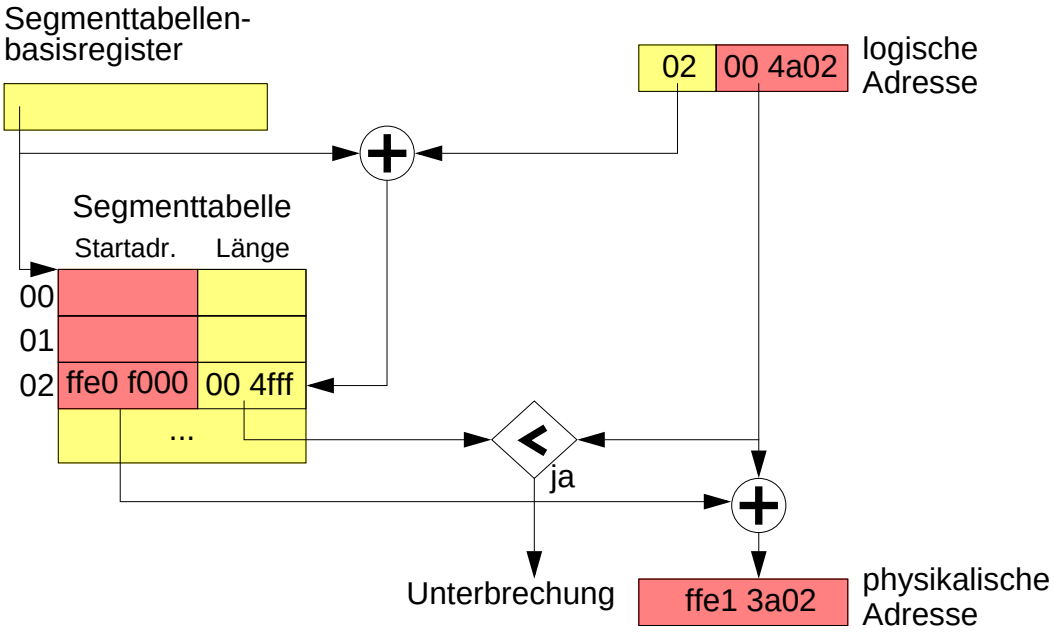
- Hardwareunterstützung: Umsetzung logischer in physikalische Adressen
  - Prozesse erhalten einen logischen Adressraum



Das Segment des logischen Adressraums kann an jeder beliebige Stelle im physikalischen Adressraum liegen.

2.4 Segmentierung (2)

- Realisierung mit Übersetzungstabelle



2.4 Segmentierung (3)

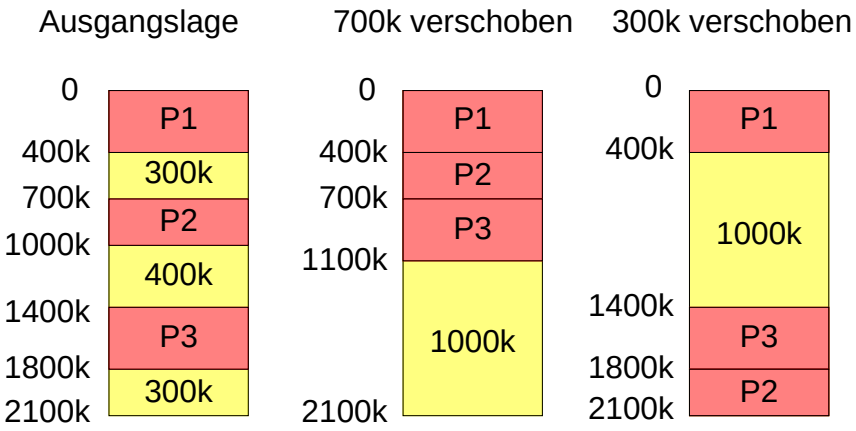
- Hardware wird MMU *(Memory Management Unit)* genannt
- Schutz vor Segmentübertretung
  - Unterbrechung zeigt Speicherverletzung an
  - Programme und Betriebssystem voreinander geschützt
- Prozessumschaltung durch Austausch der Segmentbasis
  - jeder Prozess hat eigene Übersetzungstabelle
- Ein- und Auslagerung vereinfacht
  - nach Einlagerung an beliebige Stelle muss lediglich die Übersetzungstabelle angepasst werden
- Gemeinsame Segmente möglich
  - Befehlssegmente
  - Datensegmente (*Shared Memory*)

2.4 Segmentierung (4)

- Zugriffsschutz einfach integrierbar
  - z.B. Rechte zum Lesen, Schreiben und Ausführen von Befehlen, die von der MMU geprüft werden
- Fragmentierung des Speichers durch häufiges Ein- und Auslagern
  - es entstehen kleine, nicht nutzbare Lücken
- Kompaktifizieren
  - Segmente werden verschoben, um Lücken zu schließen; Segmenttabelle wird jeweils angepasst
- lange E/A Zeiten für Ein- und Auslagerung
  - nicht alle Teile eines Segments werden gleich häufig genutzt

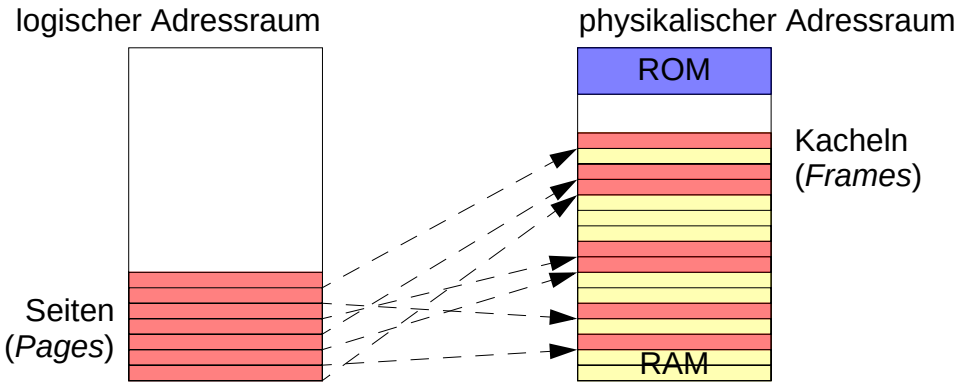
2.5 Kompaktifizieren

- Verschieben von Segmenten
  - Erzeugen von weniger aber größeren Lücken
  - Verringern des Verschnitts
  - aufwendige Operation, abhängig von der Größe der verschobenen Segmente



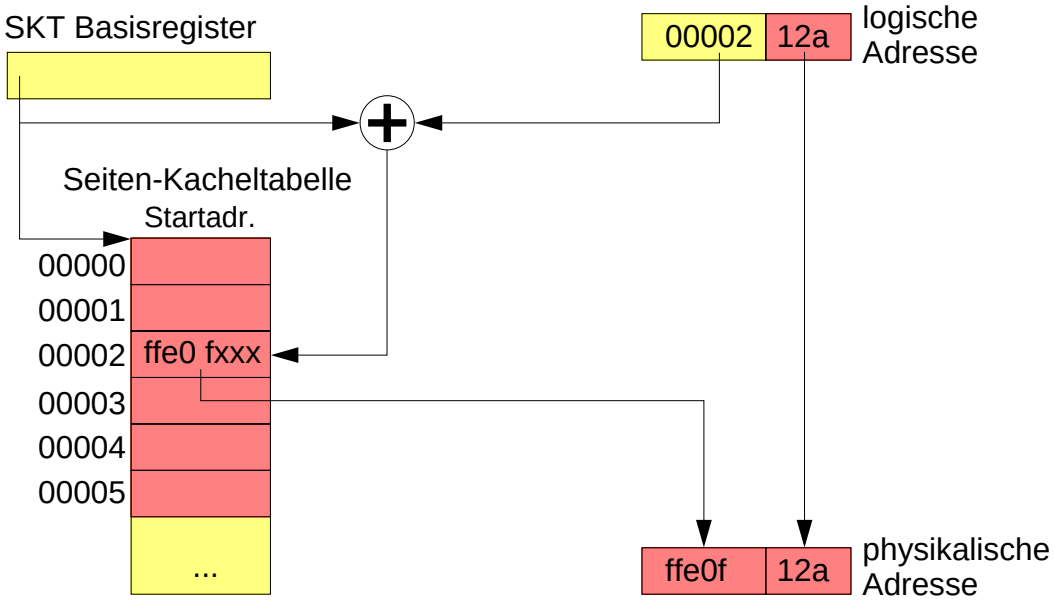
3 Seitenadressierung (*Paging*)

- Einteilung des logischen Adressraums in gleichgroße Seiten, die an beliebigen Stellen im physikalischen Adressraum liegen können
  - ◆ Lösung des Fragmentierungsproblem
  - ◆ keine Kompaktifizierung mehr nötig
  - ◆ Vereinfacht Speicherbelegung und Ein-, Auslagerungen



3.1 MMU mit Seiten-Kacheltabelle

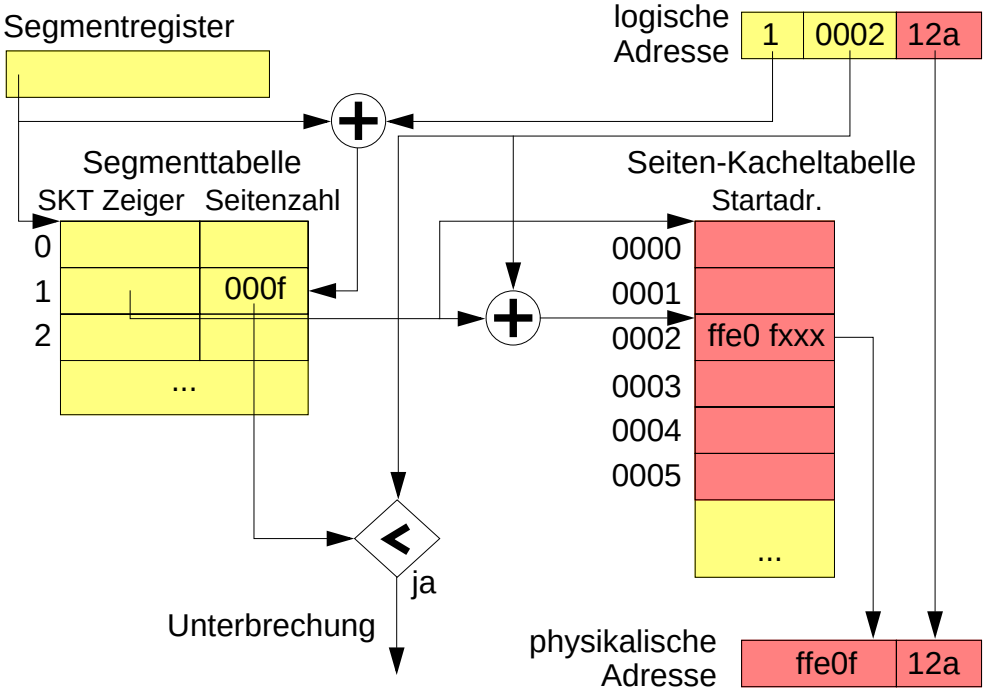
- Tabelle setzt Seiten in Kacheln um



3.1 MMU mit Seiten-Kacheltabelle (2)

- ▲ Seitenadressierung erzeugt internen Verschnitt
  - ◆ letzte Seite eventuell nicht vollständig genutzt
- ■ Seitengröße
  - ◆ kleine Seiten verringern internen Verschnitt, vergrößern aber die Seiten-Kacheltabelle (und umgekehrt)
  - ◆ übliche Größen: 512 Bytes — 8192 Bytes
- ▲ große Tabelle, die im Speicher gehalten werden muss
- ▲ viele implizite Speicherzugriffe nötig
- ▲ nur ein „Segment“ pro Kontext
- ★ Kombination mit Segmentierung

3.2 Segmentierung und Seitenadressierung



3.2 Segmentierung und Seitenadressierung (2)

- ▲ noch mehr implizite Speicherzugriffe
- ▲ große Tabellen im Speicher
- ★ Mehrstufige Seitenadressierung mit Ein- und Auslagerung

3.3 Ein- und Auslagerung von Seiten

- Es ist nicht nötig ein gesamtes Segment aus- bzw. einzulagern
  - ◆ Seiten können einzeln ein- und ausgelagert werden
- Hardware-Unterstützung

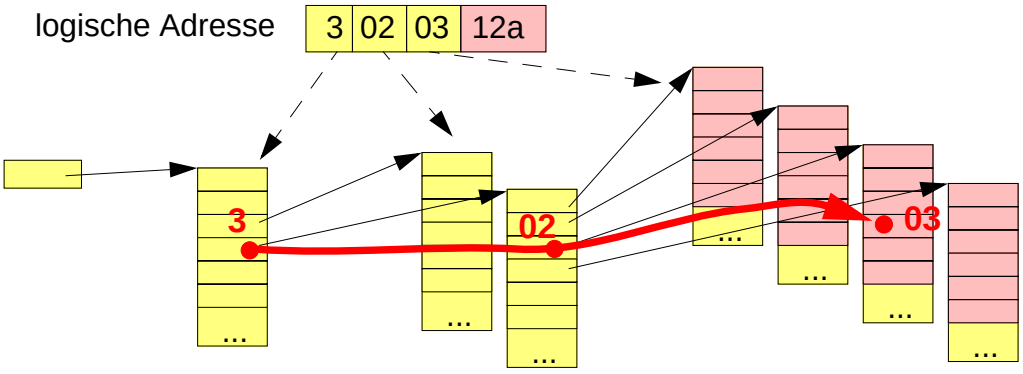
Seiten-Kacheltabelle

	Startadr.	Präsenzbit
0000		
0001		
0002	ffe0 fxxx	X
	...	

- ◆ Ist das Präsenzbit gesetzt, bleibt alles wie bisher.
- ◆ Ist das Präsenzbit gelöscht, wird eine Unterbrechung ausgelöst (*Page fault*).
- ◆ Die Unterbrechungsbehandlung kann nun für das Laden der Seite vom Hintergrundspeicher sorgen und den Speicherzugriff danach wiederholen (benötigt HW Support in der CPU).

3.4 Mehrstufige Seitenadressierung

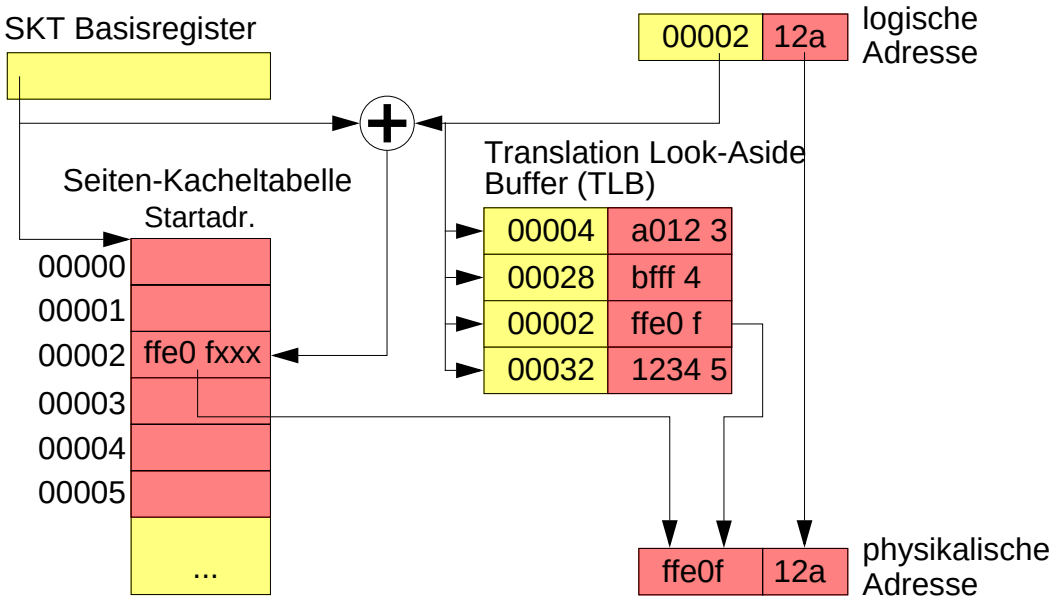
- Beispiel: zweifach indirekte Seitenadressierung



- Präsenzbit auch für jeden Eintrag in den höheren Stufen
  - Tabellen werden aus- und einlagerbar
- Noch mehr implizite Speicherzugriffe

3.5 Translation Look-Aside Buffer

- Schneller Registersatz wird konsultiert bevor auf die SKT zugegriffen wird:

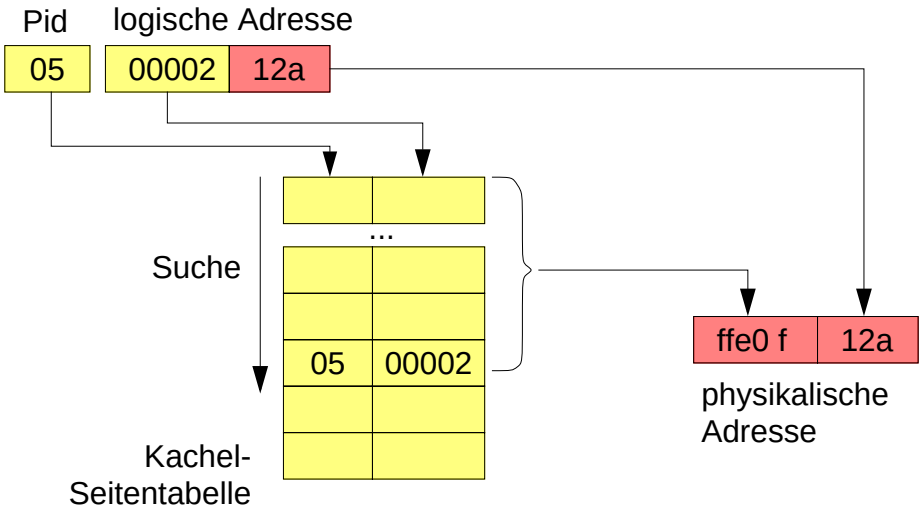


3.5 Translation Look-Aside Buffer (2)

- Schneller Zugriff auf Seitenabbildung, falls Information im voll-assoziativen Speicher des TLB
  - keine impliziten Speicherzugriffe nötig
- Bei Kontextwechseln muss TLB gelöscht werden (*Flush*)
- Bei Zugriffen auf eine nicht im TLB enthaltene Seite wird die entsprechende Zugriffsinformation in den TLB eingetragen
  - Ein alter Eintrag muss zur Ersetzung ausgesucht werden
- TLB Größe
  - Pentium: Daten TLB = 64, Code TLB = 32, Seitengröße 4K
  - Sparc V9: Daten TLB = 64, Code TLB = 64, Seitengröße 8K
  - Größere TLBs bei den üblichen Taktraten zur Zeit nicht möglich

3.6 Invertierte Seiten-Kacheltabelle

- Zum Umsetzen der Adressen nur Abbildung der belegten Kacheln nötig
  - eine Tabelle, die zu jeder Kachel die Seitenabbildung hält



3.6 Invertierte Seiten-Kacheltabelle (2)

- Vorteile
  - wenig Platz zur Speicherung der Abbildung notwendig
  - Tabelle kann immer im Hauptspeicher gehalten werden
- Nachteile
  - prozesslokale SKT zusätzlich nötig für Seiten, die ausgelagert sind
    - diese können aber ausgelagert werden
  - Suche in der KST ist aufwendig
    - Einsatz von Assoziativspeichern und Hashfunktionen

3.7 Systemaufruf

- Ermitteln der Seitengröße des Betriebssystems

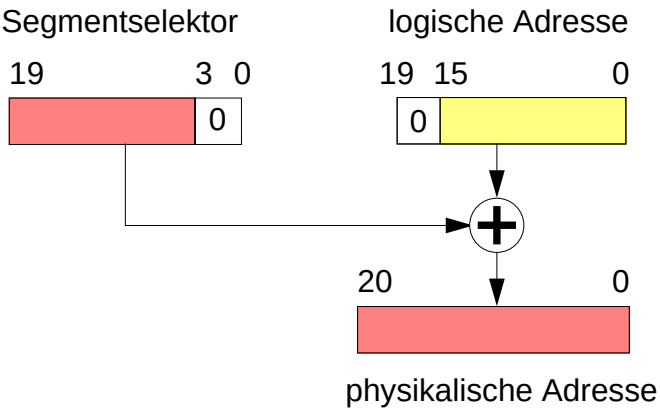
```
int getpagesize(void);
```


4 Fallstudie: Pentium

- Physikalische Adresse
  - 32 bit breit
- Segmente
  - CS – Codesegment: enthält Instruktionen
  - DS – Datensegment
  - SS – Stacksegment
  - ES, FS, GS – zusätzliche Segmente
  - Befehle beziehen sich auf eines oder mehrere der Segmente
- Segmentadressierung
  - Segmentselektor zur Auswahl eines Segments:
    16 bit bezeichnen das Segment

4.1 Real Mode Adressierung

- Adressgenerierung im Real Mode
  - 16 bit breiter Segmentselektor wird als 20 bit breite Adresse interpretiert und auf die logische Adresse addiert



4.2 Protected Mode Adressierung

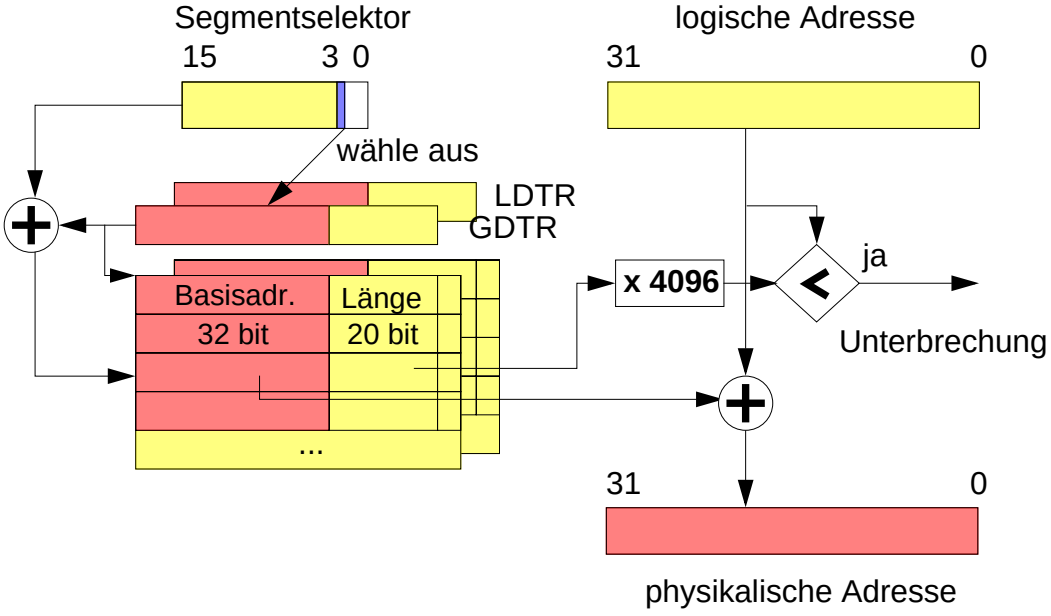
- Vier Betriebsmodi (Stufen von Privilegien)
  - Stufe 0: höchste Privilegien (privilegierte Befehle, etc.): BS Kern
  - Stufe 1: BS Treiber
  - Stufe 2: BS Erweiterungen
  - Stufe 3: Benutzerprogramme
  - Speicherverwaltung kann nur in Stufe 0 konfiguriert werden
- Segmentselektoren enthalten Privlegierungsstufe
  - Stufe des Codesegments entscheidet über Zugriffserlaubnis
- Segmentselektoren werden als Indizes interpretiert
  - Tabellen von Segmentdeskriptoren
    - Globale Deskriptor Tabelle
    - Lokale Deskriptor Tabelle

4.2 Protected Mode Adressierung (2)

- Deskriptortabelle
  - ◆ enthält bis zu 8192 Segmentdeskriptoren
  - ◆ Inhalt des Segmentdeskriptors:
    - physikalische Basisadresse
    - Längenangabe
    - Granularität (Angaben für Bytes oder Seiten)
    - Präsenzbit
    - Privilegierungsstufe
  - ◆ globale Deskriptortabelle für alle Prozesse zugänglich (Register GDTR)
  - ◆ lokale Deskriptortabelle pro Prozess möglich
    (Register LDTR gehört zum Prozesskontext)

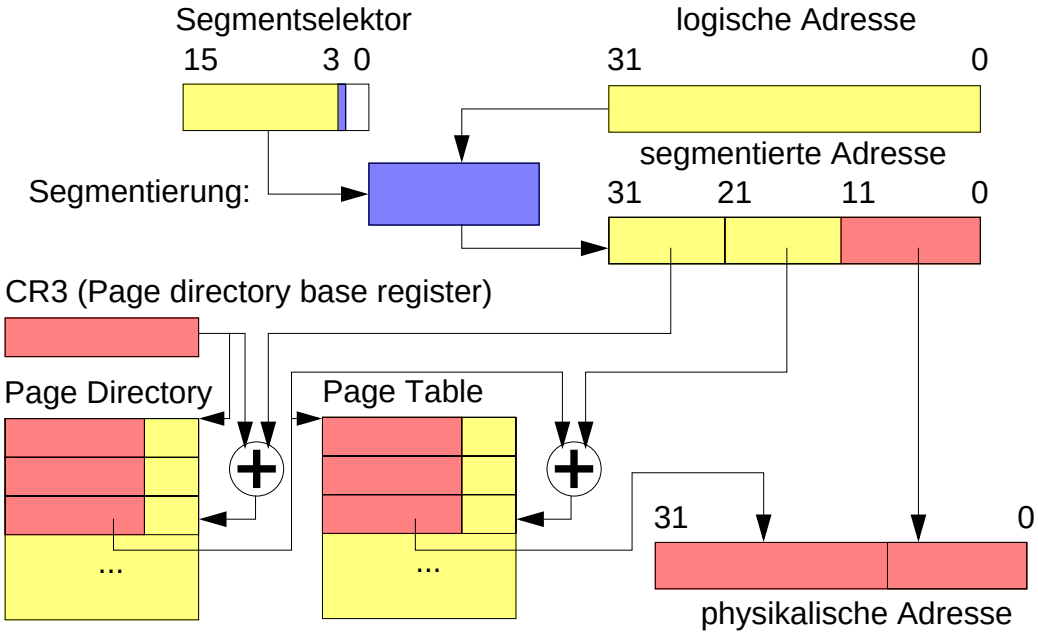
4.3 Adressberechnung bei Segmentierung

- Verwendung der Protected mode Adressierung



4.4 Adressberechnung bei Paging

- Seitenadressierung wird der Segmentierung nachgeschaltet

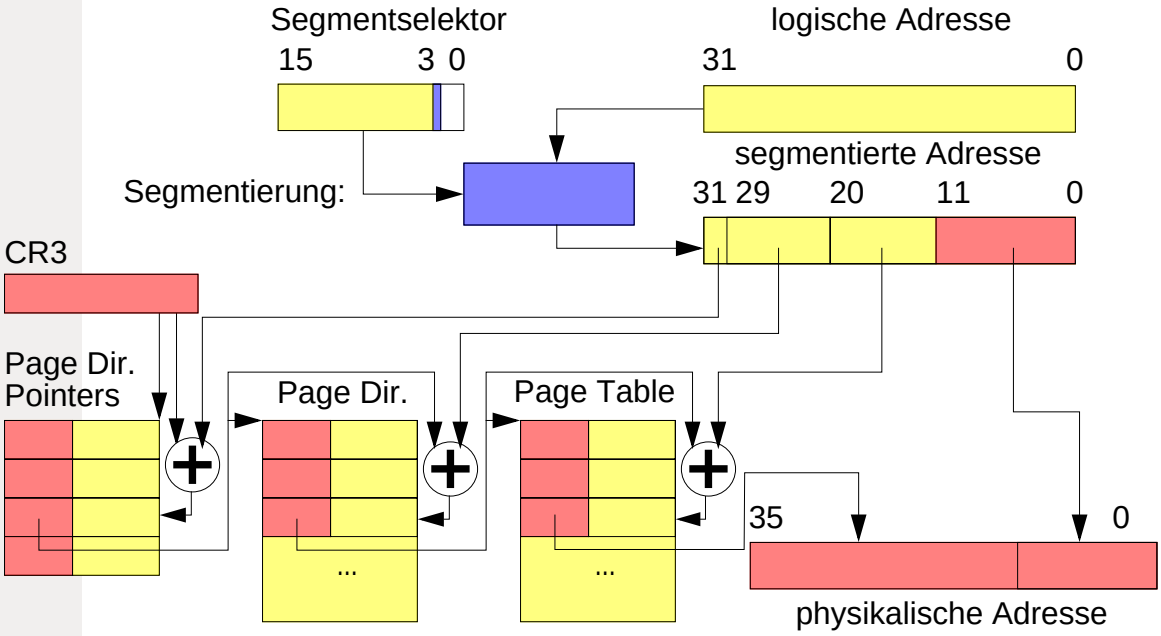


4.4 Adressberechnung bei Paging (2)

- Zweistufige Seitenadressierung
  - Directory — Page table
  - Seitengröße fest auf 4096 Bytes
- Inhalt des Seitendeskriptor
  - Kacheladresse
  - Dirty Bit: Seite wurde beschrieben
  - Access Bit: Seite wurde gelesen oder geschrieben
  - Schreibschutz: Seite nur lesbar
  - Präsenzbit: Seite ausgelagert (31 Bits für BS-Informationen nutzbar)
  - Kontrolle des Prozessorcaches
- Getrennte TLBs für Codesegment und Datensegmente
  - 64 Einträge für Datenseiten; 32 Einträge für Codeseiten

4.5 Physical-Address-Extension (PAE)

- Nachgeschaltete dreistufige Seitenadressierung



4.5 Physical-Address-Extension (PAE) (2)

- Ab Pentium Pro
  - ◆ vierelementige Tabelle von Page-Directory-Pointers
  - ◆ 24 statt 20 Bit breite physikalische Adressumsetzung für den Seitenanfang
  - ◆ 64 Bit statt 32 Bit breite Tabelleneinträge
  - ◆ spezieller Chipsatz erforderlich
- Adressierbarer Hauptspeicher von 64 GByte

5 Gemeinsamer Speicher (*Shared Memory*)

- Speicher, der mehreren Prozessen zur Verfügung steht
  - gemeinsame Segmente (gleiche Einträge in verschiedenen Segmenttabellen)
  - gemeinsame Seiten (gleiche Einträge in verschiedenen SKTs)
  - gemeinsame Seitenbereiche (gemeinsames Nutzen einer SKT bei mehrstufigen Tabellen)
- Gemeinsamer Speicher wird beispielsweise benutzt für
  - Kommunikation zwischen Prozessen
  - gemeinsame Befehlssegmente

5 Gemeinsamer Speicher (2)

- Systemaufrufe unter Solaris 2.5
  - Erzeugen bzw. Holen eines gemeinsamen Speichersegments

```
int shmget( key_t key, int size, int shmflg );
```

  - Einblenden und Ausblenden des Segments in den Speicher

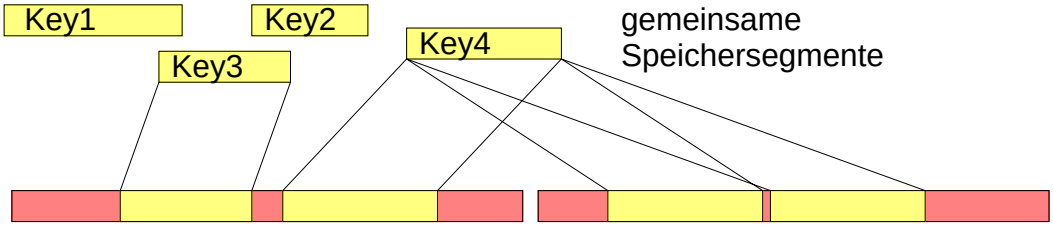
```
void *shmat( int shmid, void *shmaddr, int shmflg );
int shmdt( void *shmaddr );
```

  - Kontrolloperation

```
int shmctl( int shmid, int cmd, struct shmid_ds *buf );
```

5 Gemeinsamer Speicher (3)

- Prinzip der **shm*** Operationen




```
id1= shmget( Key3, ...);
shmat( id1, NULL, ...);
id2= shmget( Key4, ...);
shmat( id2, NULL, ...);
```

```
id1= shmget( Key4, ...);
shmat( id1, NULL, ...);
shmat( id1, NULL, ...);
```


5 Gemeinsamer Speicher (4)

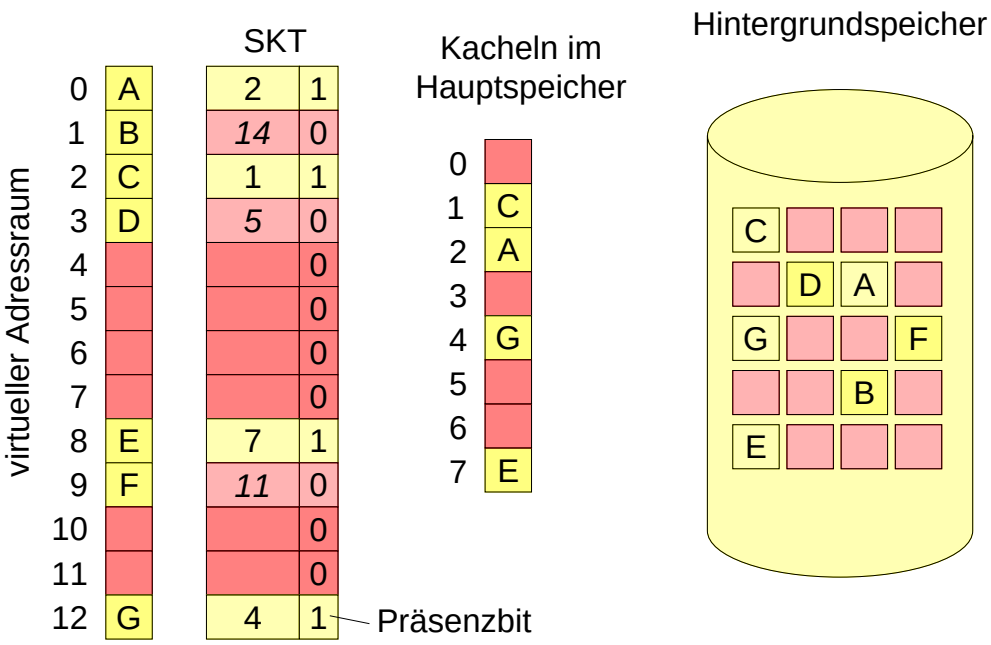
- Verwendung des Keys
  - Alle Prozesse, die auf ein Speichersegment zugreifen wollen, müssen den Key kennen
  - Keys sind eindeutig innerhalb eines (Betriebs-)Systems
  - Ist ein Key bereits vergeben, kann kein Segment mit gleichem Key erzeugt werden
  - Ist ein Key bekannt, kann auf das Segment zugegriffen werden
    - gesetzte Zugriffsberechtigungen werden allerdings beachtet
  - Es können Segmente ohne Key erzeugt werden (private Segmente)
- Keys werden benutzt für:
  - Queues
  - Semaphore
  - Shared memory segments

6 Virtueller Speicher

- Entkoppelung des Speicherbedarfs vom verfügbaren Hauptspeicher
  - ◆ Prozesse benötigen nicht alle Speicherstellen gleich häufig
    - bestimmte Befehle werden selten oder gar nicht benutzt (z.B. Fehlerbehandlungen)
    - bestimmte Datenstrukturen werden nicht voll belegt
  - ◆ Prozesse benötigen evtl. mehr Speicher als Hauptspeicher vorhanden
- Idee
  - ◆ Vortäuschen eines großen Hauptspeichers
  - ◆ Einblenden benötiger Speicherbereiche
  - ◆ Abfangen von Zugriffen auf nicht-eingeblendete Bereiche
  - ◆ Bereitstellen der benötigen Bereiche auf Anforderung
  - ◆ Auslagern nicht-benötigter Bereiche

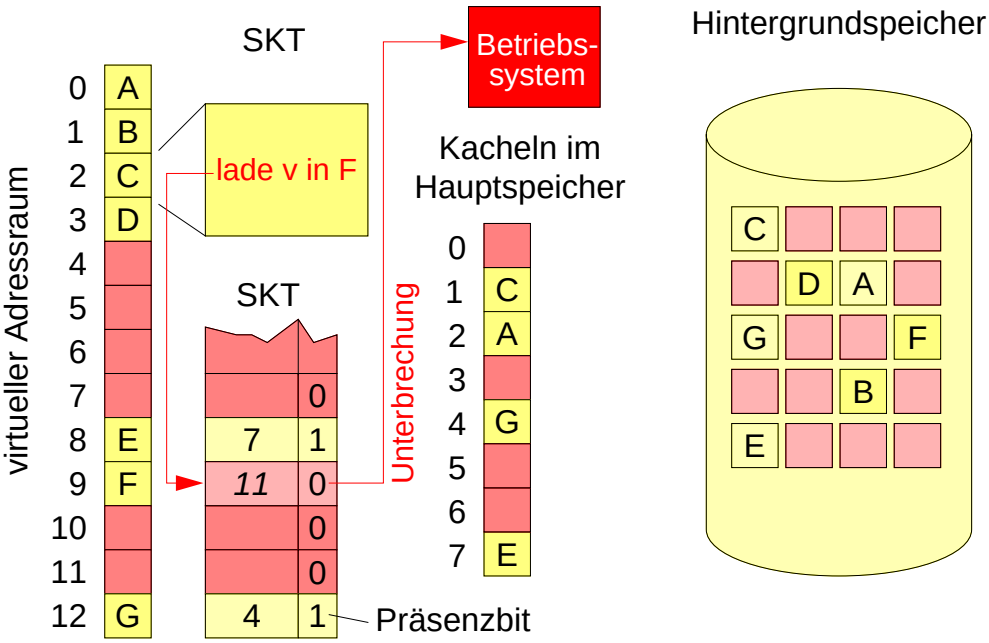
6.1 Demand Paging

- Bereitstellen von Seiten auf Anforderung



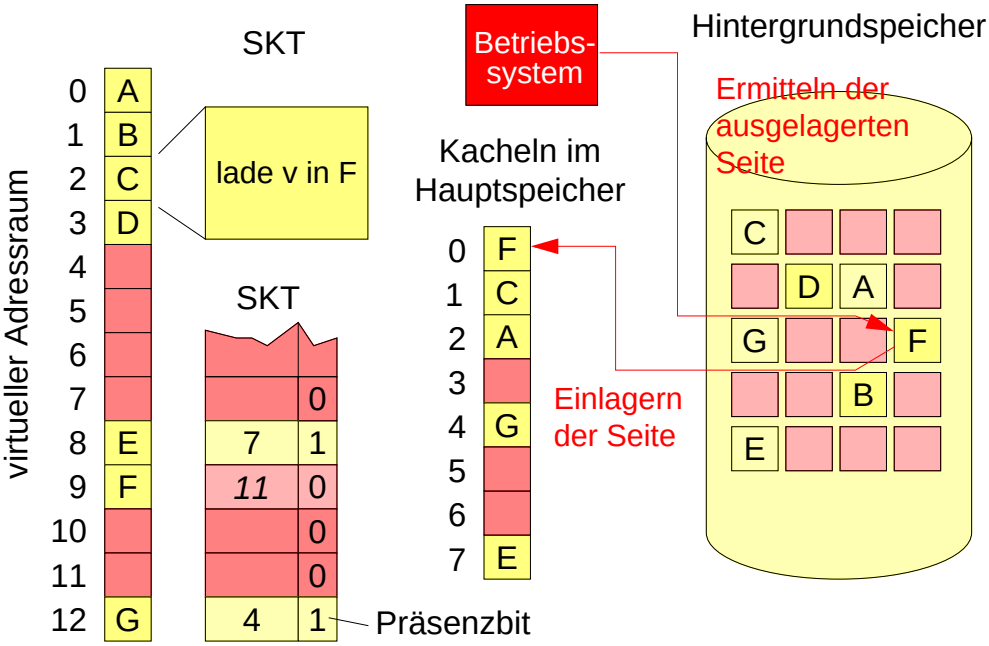
6.1 Demand Paging (2)

- Reaktion auf Seitenfehler



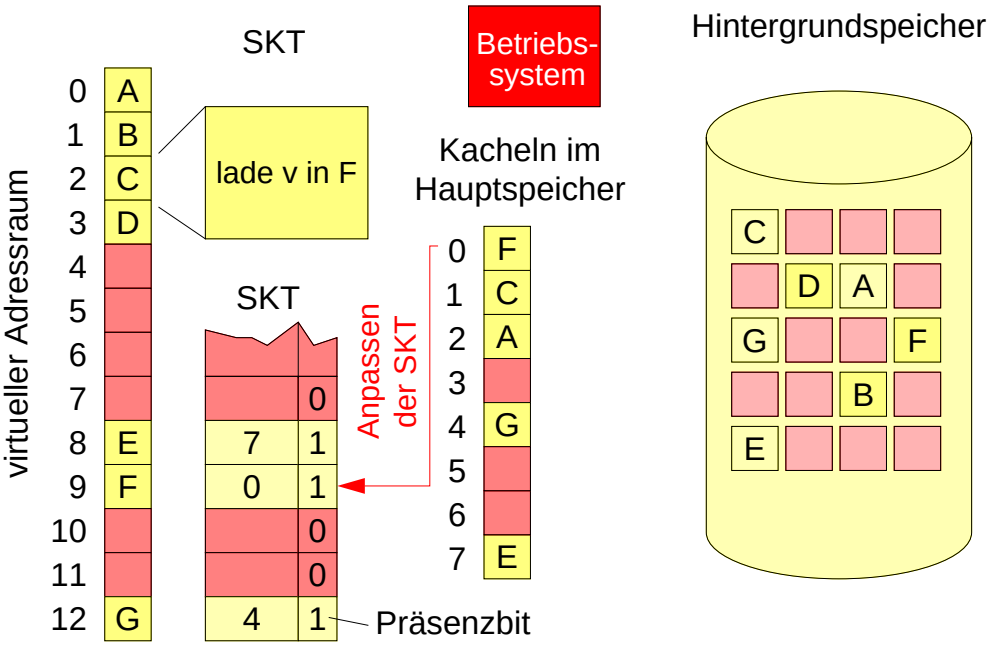
6.1 Demand Paging (3)

- Reaktion auf Seitenfehler



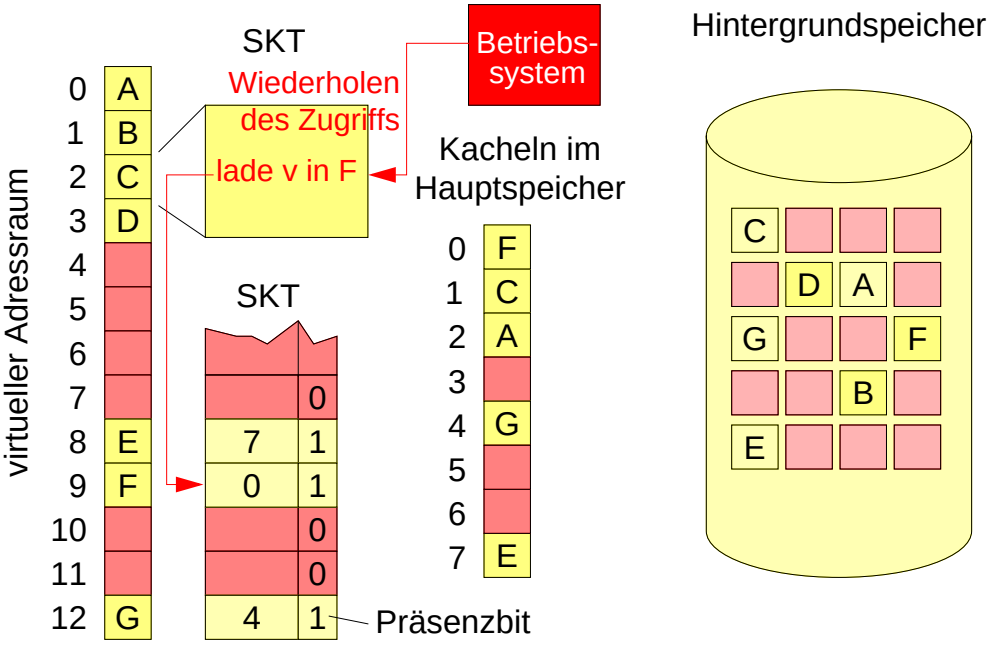
6.1 Demand Paging (4)

- Reaktion auf Seitenfehler



6.1 Demand Paging (5)

- Reaktion auf Seitenfehler



6.1 Demand Paging (6)

- Performanz von Demand paging
  - Keine Seitenfehler
    - effektive Zugriffszeit zw. 10 und 200 Nanosekunden
  - Mit Seitenfehler
    - p sei Wahrscheinlichkeit für Seitenfehler; p nahe Null
    - Annahme: Zeit zum Einlagern einer Seite vom Hintergrundspeicher gleich 25 Millisekunden (8 ms Latenz, 15 ms Positionierzeit, 1 ms Übertragungszeit)
    - Annahme: normale Zugriffszeit 100 ns
    - Effektive Zugriffszeit:
      $(1 - p) \times 100 + p \times 25000000 = 100 + 24999900 \times p$
- Seitenfehler müssen so niedrig wie möglich gehalten werden
- Abwandlung: *Demand zero* für nicht initialisierte Daten

6.2 Seitenersetzung

- Was tun, wenn keine freie Kachel vorhanden?
  - Eine Seite muss verdrängt werden, um Platz für neue Seite zu schaffen!
  - Auswahl von Seiten, die nicht geändert wurden (*Dirty bit* in der SKT)
  - Verdrängung erfordert Auslagerung, falls Seite geändert wurde
- Vorgang:
  - Seitenfehler *(Page fault)*: Unterbrechung
  - Auslagern einer Seite, falls keine freie Kachel verfügbar
  - Einlagern der benötigten Seite
  - Wiederholung des Zugriffs
- Problem
  - Welche Seite soll ausgewählt werden?

7 Ersetzungsstrategien

- Betrachtung von Ersetzungsstrategien und deren Wirkung auf Referenzfolgen
- Referenzfolge
  - ◆ Folge von Seitennummern, die das Speicherzugriffsverhalten eines Prozesses abbildet
  - ◆ Ermittlung von Referenzfolgen z.B. durch Aufzeichnung der zugegriffenen Adressen
    - Reduktion der aufgezeichneten Sequenz auf Seitennummern
    - Zusammenfassung von unmittelbar hintereinanderstehenden Zugriffen auf die gleiche Seite
  - ◆ Beispiel für eine Referenzfolge: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

7.1 First-In, First-Out

- Älteste Seite wird ersetzt
- Notwendige Zustände:
  - ◆ Alter bzw. Einlagerungszeitpunkt für jede Kachel
- Ablauf der Ersetzungen (9 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	5	5	5
	Kachel 2		2	2	2	1	1	1	1	1	3	3	3
	Kachel 3			3	3	3	2	2	2	2	2	4	4
Kontrollzustände (Alter pro Kachel)	Kachel 1	0	1	2	0	1	2	0	1	2	3	4	5
	Kachel 2	>	0	1	2	0	1	2	3	4	0	1	2
	Kachel 3	>	>	0	1	2	0	1	2	3	4	0	1

7.1 First-In, First-Out (2)

- Größerer Hauptspeicher mit 4 Kacheln (10 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	5	5	5	5	4	4
	Kachel 2		2	2	2	2	2	2	1	1	1	1	5
	Kachel 3			3	3	3	3	3	3	2	2	2	2
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontrollzustände (Alter pro Kachel)	Kachel 1	0	1	2	3	4	5	0	1	2	3	0	1
	Kachel 2	>	0	1	2	3	4	5	0	1	2	3	0
	Kachel 3	>	>	0	1	2	3	4	5	0	1	2	3
	Kachel 4	>	>	>	0	1	2	3	4	5	0	1	2

- FIFO Anomalie (Belady's Anomalie, 1969)

7.2 Optimale Ersetzungsstrategie

- Vorwärtsabstand
  - ◆ Zeitdauer bis zum nächsten Zugriff auf die entsprechende Seite
- Strategie B_0 (OPT oder MIN) ist optimal (bei fester Kachelmenge): minimale Anzahl von Einlagerungen/Ersetzungen (hier 7)
  - ◆ „Ersetze immer die Seite mit dem größten Vorwärtsabstand!“

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	3	4	4
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	4	4	4	5	5	5	5	5	5
Kontrollzustände (Vorwärts-abstand)	Kachel 1	4	3	2	1	3	2	1	>	>	>	>	>
	Kachel 2	>	4	3	2	1	3	2	1	>	>	>	>
	Kachel 3	>	>	7	7	6	5	5	4	3	2	1	>

7.2 Optimale Ersetzungsstrategie (2)

- Vergrößerung des Hauptspeichers (4 Kacheln): 6 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	1	4	4
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	3	3	3	3	3	3	3	3	3
	Kachel 4				4	4	4	5	5	5	5	5	5
Kontrollzustände (Vorwärts-abstand)	Kachel 1	4	3	2	1	3	2	1	>	>	>	>	>
	Kachel 2	>	4	3	2	1	3	2	1	>	>	>	>
	Kachel 3	>	>	7	6	5	4	3	2	1	>	>	>
	Kachel 4	>	>	>	7	6	5	5	4	3	2	1	>

  - keine Anomalie

7.2 Optimale Ersetzungsstrategie (3)

- Implementierung von B_0 nahezu unmöglich
  - Referenzfolge müsste vorher bekannt sein
  - B_0 meist nur zum Vergleich von Strategien brauchbar
- Suche nach Strategien, die möglichst nahe an B_0 kommen
  - z.B. *Least recently used* (LRU)

7.3 Least Recently Used (LRU)

- Rückwärtsabstand
  - ◆ Zeitdauer, seit dem letzten Zugriff auf die Seite
- LRU Strategie (10 Einlagerungen)
  - ◆ „Ersetze die Seite mit dem größten Rückwärtsabstand !“

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	3	3	3
	Kachel 2		2	2	2	1	1	1	1	1	1	4	4
	Kachel 3			3	3	3	2	2	2	2	2	2	5
Kontrollzustände (Rückwärts-abstand)	Kachel 1	0	1	2	0	1	2	0	1	2	0	1	2
	Kachel 2	>	0	1	2	0	1	2	0	1	2	0	1
	Kachel 3	>	>	0	1	2	0	1	2	0	1	2	0

7.3 Least Recently Used (2)

- Vergrößerung des Hauptspeichers (4 Kacheln): 8 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	1	1	1	1	1	5
	Kachel 2		2	2	2	2	2	2	2	2	2	2	2
	Kachel 3			3	3	3	3	5	5	5	5	4	4
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontrollzustände (Rückwärts-abstand)	Kachel 1	0	1	2	3	0	1	2	0	1	2	3	0
	Kachel 2	>	0	1	2	3	0	1	2	0	1	2	3
	Kachel 3	>	>	0	1	2	3	0	1	2	3	0	1
	Kachel 4	>	>	>	0	1	2	3	4	5	0	1	2

7.3 Least Recently Used (3)

- Keine Anomalie
  - Allgemein gilt: Es gibt eine Klasse von Algorithmen (Stack-Algorithmen), bei denen keine Anomalie auftritt:
    - Bei Stack-Algorithmen ist bei n Kacheln zu jedem Zeitpunkt eine Untermenge der Seiten eingelagert, die bei n+1 Kacheln zum gleichen Zeitpunkt eingelagert wären!
    - LRU: Es sind immer die letzten n benutzten Seiten eingelagert
    - B_0: Es sind die n bereits benutzten Seiten eingelagert, die als nächstes zugegriffen werden
- Problem
  - Implementierung von LRU nicht ohne Hardwareunterstützung möglich
  - Es muss jeder Speicherzugriff berücksichtigt werden

7.3 Least Recently Used (4)

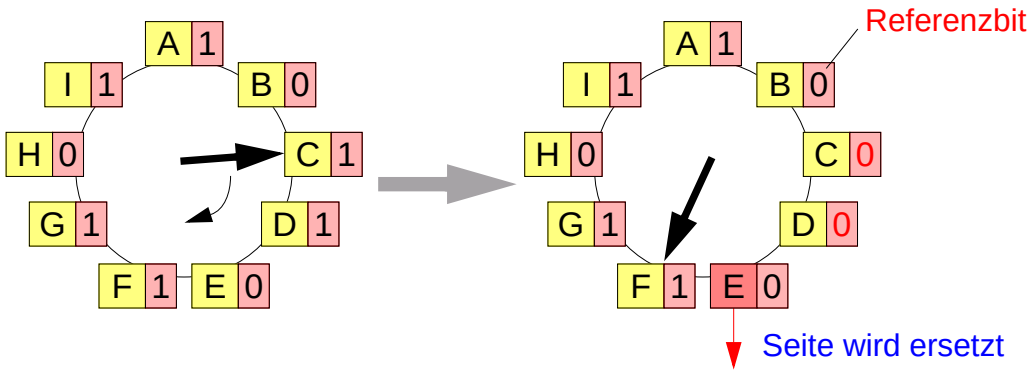
- Hardwareunterstützung durch Zähler
  - CPU besitzt einen Zähler, der bei jedem Speicherzugriff erhöht wird (inkrementiert wird)
  - bei jedem Zugriff wird der aktuelle Zählerwert in den jeweiligen Seitendeskriptor geschrieben
  - Auswahl der Seite mit dem kleinsten Zählerstand
- Aufwendige Implementierung
  - viele zusätzliche Speicherzugriffe

7.4 Second Chance (Clock)

- Einsatz von Referenzbits
  - Referenzbit im Seitendeskriptor wird automatisch durch Hardware gesetzt, wenn die Seite zugegriffen wird
    - einfacher zu implementieren
    - weniger zusätzliche Speicherzugriffe
    - moderne Prozessoren bzw. MMUs unterstützen Referenzbits (z.B. Pentium: *Access bit*)
- Ziel: Annäherung von LRU
  - das Referenzbit wird zunächst auf 0 gesetzt
  - wird eine Opferseite gesucht, so werden die Kacheln reihum inspiziert
  - ist das Referenzbit 1, so wird es auf 0 gesetzt (zweite Chance)
  - ist das Referenzbit 0, so wird die Seite ersetzt

7.4 Second Chance (2)

- Implementierung mit umlaufendem Zeiger *(Clock)*



  - an der Zeigerposition wird Referenzbit getestet
    - falls Referenzbit eins, wird Bit gelöscht
    - falls Referenzbit gleich Null, wurde ersetzbare Seite gefunden
    - Zeiger wird weitergestellt; falls keine Seite gefunden: Wiederholung
  - falls alle Referenzbits auf 1 stehen, wird Second chance zu FIFO

7.4 Second Chance (3)

- Ablauf bei drei Kacheln (9 Einlagerungen)

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	4	4	4	5	5	5	5	5	5
	Kachel 2		2	2	2	1	1	1	1	1	3	3	3
	Kachel 3			3	3	3	2	2	2	2	2	4	4
Kontroll-zustände (Referenzbits)	Kachel 1	1	1	**1**	1	1	**1**	1	1	1	0	**0**	**1**
	Kachel 2	**0**	1	1	**0**	1	1	**0**	**1**	**1**	1	1	1
	Kachel 3	0	**0**	1	0	**0**	1	0	0	1	**0**	1	1
	Umlaufzeiger	2	3	1	2	3	1	2	2	2	3	1	1

7.4 Second Chance (4)

- Vergrößerung des Hauptspeichers (4 Kacheln): 10 Einlagerungen

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
Hauptspeicher	Kachel 1	1	1	1	1	1	1	5	5	5	5	4	4
	Kachel 2		2	2	2	2	2	2	1	1	1	1	5
	Kachel 3			3	3	3	3	3	3	2	2	2	2
	Kachel 4				4	4	4	4	4	4	3	3	3
Kontroll-zustände (Refernzbits)	Kachel 1	1	1	1	**1**	**1**	**1**	1	1	1	**1**	1	1
	Kachel 2	**0**	1	1	1	1	1	**0**	1	1	1	**0**	1
	Kachel 3	0	**0**	1	1	1	1	0	**0**	1	1	0	**0**
	Kachel 4	0	0	**0**	1	1	1	0	0	**0**	1	0	0
	Umlaufzeiger	2	3	4	1	1	1	2	3	4	1	2	3

7.4 Second Chance (5)

- Second chance zeigt FIFO Anomalie
  - Wenn alle Referenzbits gleich 1, wird nach FIFO entschieden
- Erweiterung
  - Modifikationsbit kann zusätzlich berücksichtigt werden *(Dirty bit)*
  - drei Klassen: (0,0), (1,0) und (1,1) mit (Referenzbit, Modifikationsbit)
  - Suche nach der niedrigsten Klasse (Einsatz im MacOS)

7.5 Freiseitenpuffer

- Statt eine Seite zu ersetzen wird permanent eine Menge freier Seiten gehalten
  - Auslagerung geschieht im „voraus“
  - Effizienter: Ersetzungszeit besteht im Wesentlichen nur aus Einlagerungszeit
- Behalten der Seitenzuordnung auch nach der Auslagerung
  - Wird die Seite doch noch benutzt bevor sie durch eine andere ersetzt wird, kann sie mit hoher Effizienz wiederverwendet werden.
  - Seite wird aus Freiseitepuffer ausgetragen und wieder dem entsprechenden Prozess zugeordnet.

7.6 Seitenanforderung

- Problem: Zuordnung der Kacheln zu mehreren Prozessen
- Begrenzungen
  - Maximale Seitenmenge: begrenzt durch Anzahl der Kacheln
  - Minimale Seitenmenge: abhängig von der Prozessorarchitektur
    - Mindestens die Anzahl von Seiten nötig, die theoretisch bei einem Maschinenbefehl benötigt werden
      (z.B. zwei Seiten für den Befehl, vier Seiten für die adressierten Daten)
- Gleiche Zuordnung
  - Anzahl der Prozesse bestimmt die Kachelmenge, die ein Prozess bekommt
- Größenabhängige Zuordnung
  - Größe des Programms fließt in die zugeteilte Kachelmenge ein

7.6 Seitenanforderung

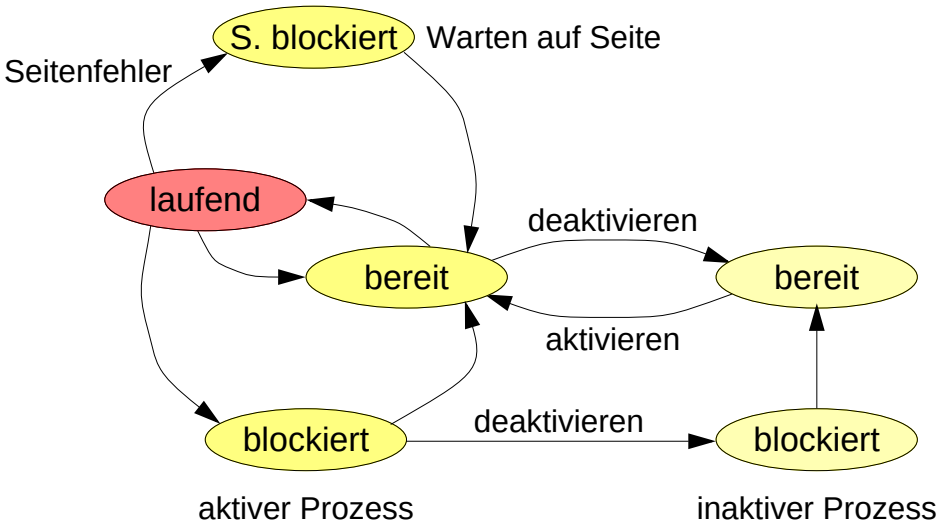
- Globale und lokale Anforderung von Seiten
  - lokal: Prozess ersetzt nur immer seine eigenen Seiten
    - Seitenfehler-Verhalten liegt nur in der Verantwortung des Prozesses
  - global: Prozess ersetzt auch Seiten anderer Prozesse
    - bessere Effizienz, da ungenutzte Seiten von anderen Prozessen verwendet werden können

8 Seitenflattern *(Thrashing)*

- Ausgelagerte Seite wird gleich wieder angesprochen
  - Prozess verbringt mehr Zeit mit dem Warten auf das Beheben von Seitenfehler als mit der eigentlichen Ausführung
- Ursachen
  - Prozess ist nahe am Seitenminimum
  - zu viele Prozesse gleichzeitig im System
  - schlechte Ersetzungsstrategie
- ★ Lokale Seitenanforderung behebt Thrashing zwischen Prozessen
- ★ Zuteilung einer genügend großen Zahl von Kacheln behebt Thrashing innerhalb der Prozessseiten
  - Begrenzung der Prozessanzahl

8.1 Deaktivieren von Prozessen

- Einführung von „Superzuständen“



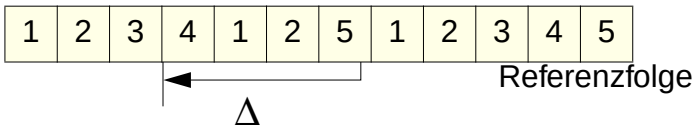
  - inaktiver Prozess benötigt keine Kacheln; Prozess ist vollständig ausgelagert (swapped out)

8.1 Deaktivieren von Prozessen (2)

- Sind zuviele Prozesse aktiv, werden welche deaktiviert
  - ◆ Kacheln teilen sich auf weniger Prozesse auf
  - ◆ Verbindung mit dem Scheduling nötig
    - Verhindern von Aushungerung
    - Erzielen kurzer Reaktionszeiten
  - ◆ guter Kandidat: Prozess mit wenigen Seiten im Hauptspeicher
    - geringe Latenz bei Wiedereinlagerung bzw. wenige Seitenfehler bei Aktivierung und Demand paging

8.2 Arbeitsmengenmodell

- Menge der Seiten, die ein Prozess wirklich braucht *(Working set)*
  - ◆ kann nur angenähert werden, da üblicherweise nicht vorhersehbar
- Annäherung durch Betrachten der letzten Δ Seiten, die angesprochen wurden
  - ◆ geeignete Wahl von Δ
    - zu groß: Überlappung von lokalen Zugriffsmustern zu klein: Arbeitsmenge enthält nicht alle nötigen Seiten



- Hinweis: $\Delta >$ Arbeitsmenge, da Seiten in der Regel mehrfach hintereinander angesprochen werden

8.2 Arbeitsmengenmodell (2)

- Beispiel: Arbeitsmengen bei verschiedenen Δ

Referenzfolge		1	2	3	4	1	2	5	1	2	3	4	5
$\Delta = 3$	Seite 1	x	x	x		x	x	x	x	x	x		
	Seite 2		x	x	x		x	x	x	x	x	x	
	Seite 3			x	x	x					x	x	x
	Seite 4				x	x	x					x	x
	Seite 5							x	x	x			x
$\Delta = 4$	Seite 1	x	x	x	x	x	x	x	x	x	x	x	
	Seite 2		x	x	x	x	x	x	x	x	x	x	x
	Seite 3			x	x	x	x				x	x	x
	Seite 4				x	x	x	x				x	x
	Seite 5							x	x	x	x		x

8.2 Arbeitsmengenmodell (3)

- Annäherung der Zugriffe durch die Zeit
  - bestimmtes Zeitintervall ist ungefähr proportional zu Anzahl von Speicherzugriffen
- Virtuelle Zeit des Prozesses muss gemessen werden
  - nur die Zeit relevant, in der der Prozess im Zustand laufend ist
  - Verwalten virtueller Uhren pro Prozess

8.3 Arbeitsmengenbestimmung mit Zeitgeber

- Annäherung der Arbeitsmenge mit
  - Referenzbit
  - Altersangabe pro Seite (Zeitintervall ohne Benutzung)
  - Timer-Interrupt (durch Zeitgeber)
- Algorithmus
  - durch regelmäßigen Interrupt wird mittels Referenzbit die Altersangabe fortgeschrieben:
    - ist Referenzbit gesetzt (Seite wurde benutzt) wird das Alter auf Null gesetzt;
    - ansonsten wird Altersangabe erhöht.
    - Es werden nur die Seiten des gerade laufenden Prozesses „gealtert“.
  - Seiten mit Alter > Δ sind nicht mehr in der Arbeitsmenge des jeweiligen Prozesses

8.3 Arbeitsmengenbestimmung mit Zeitgeber (2)

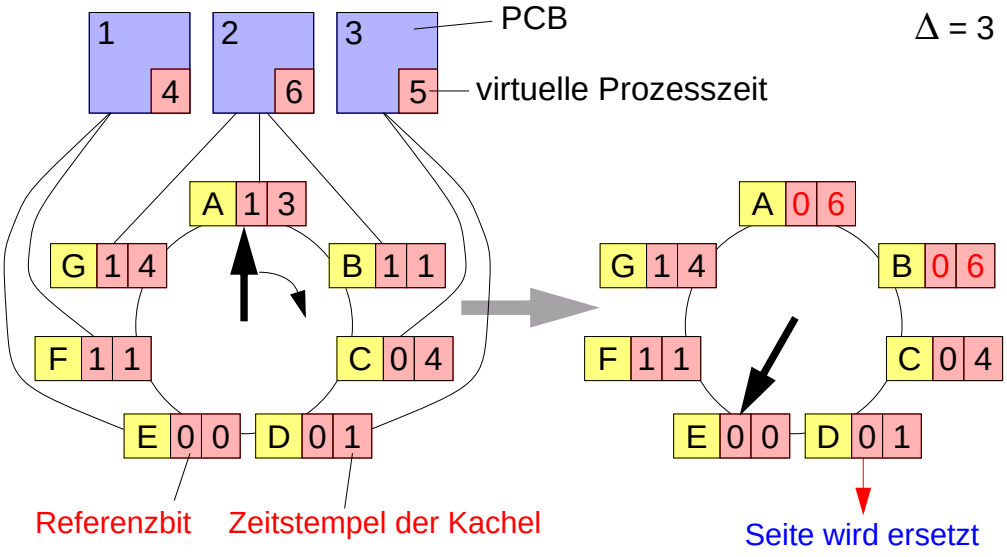
- Ungenau: System ist aber nicht empfindlich auf diese Ungenauigkeit
  - Verringerung der Zeitintervalle: höherer Aufwand, genauere Messung
- Ineffizient
  - große Menge von Seiten zu betrachten

8.4 Arbeitsmengenbestimmung mit WSClock

- Algorithmus WSClock (Working set clock)
  - arbeitet wie Clock
  - Seite wird nur dann ersetzt, wenn sie nicht zur Arbeitsmenge ihres Prozesses gehört oder der Prozess deaktiviert ist
  - Bei Zurücksetzen des Referenzbits wird die virtuelle Zeit des jeweiligen Prozesses eingetragen, die z.B. im PCB gehalten und fortgeschrieben wird
  - Bestimmung der Arbeitsmenge erfolgt durch Differenzbildung von virtueller Zeit des Prozesses und Zeitstempel in der Kachel

8.4 Arbeitsmengenbestimmung mit WSClock (2)

- WSClock Algorithmus

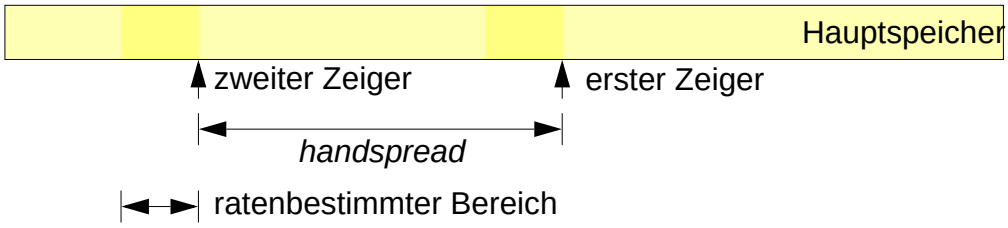


8.5 Probleme mit Arbeitsmengen

- ▲ Zuordnung zu einem Prozess nicht immer möglich
  - ◆ gemeinsam genutzte Seiten in modernen Betriebssystemen eher die Regel als die Ausnahme
    - Seiten des Codesegments
    - Shared libraries
    - Gemeinsame Seiten im Datensegment *(Shared memory)*
- ★ moderne System bestimmen meist eine globale Arbeitsmenge von Seiten

8.6 Ersetzungsstrategie bei Solaris

- ■ Prozess *pageout* arbeitet Clock-Strategie ab
  - ◆ Prozess läuft mehrmals die Sekunde (4x)
  - ◆ adaptierbare Rate: untersuchte Seiten pro Sekunde
  - ◆ statt ein Zeiger: zwei Zeiger
    - am ersten Zeiger werden Referenzbits zurückgesetzt
    - am zweiten Zeiger werden Seiten mit gelöschtem Ref.-Bit ausgewählt
    - nötig, weil sonst Zeitspanne zwischen Löschen und Auswählen zu lang wird (großer Hauptspeicher; 64 MByte entsprechen 8.192 Seiten)
    - Zeigerabstand einstellbar *(handspread)*

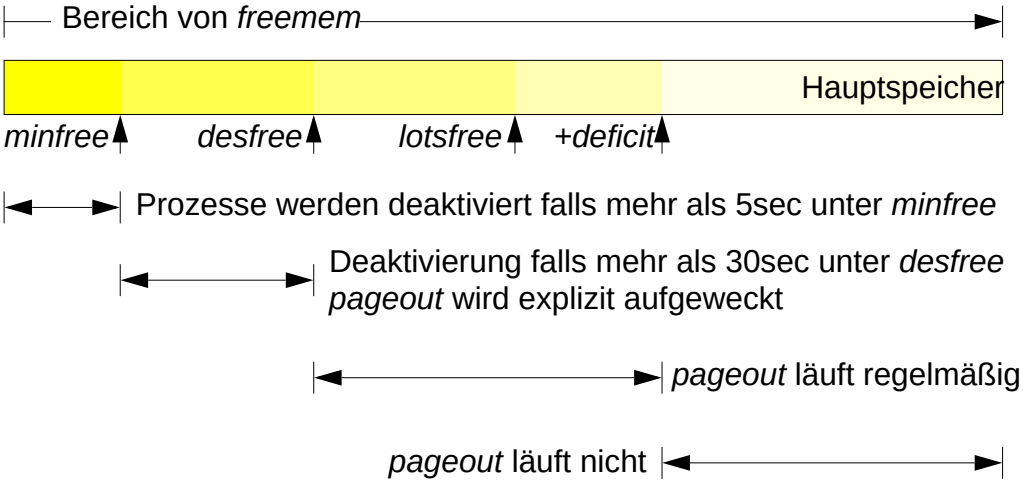


8.6 Ersetzungsstrategie bei Solaris (2)

- ermittelte Seiten werden ausgelagert (falls nötig) und
- in eine Freiliste eingehängt
- aus der Freiliste werden Kacheln für Einlagerungen angefordert
- Seitenfehler können unbenutzte Seiten aus der Freiliste wieder zurückfordern *(Minor page faults)*

8.6 Ersetzungsstrategie bei Solaris (3)

- Verhalten von *pageout* orientiert sich an Größe der Freiliste (Menge des freien Speichers)

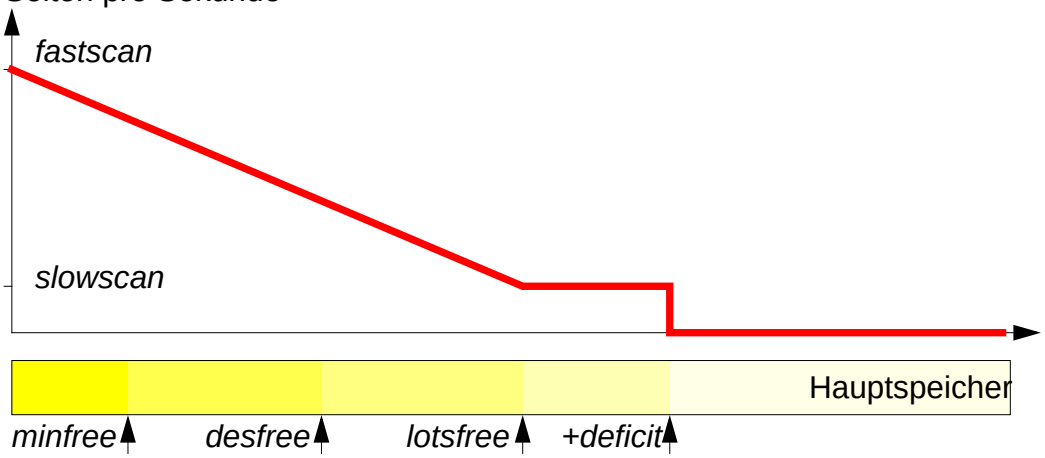


- *deficit* wird dynamisch ermittelt (0 bis *lotsfree*) und auf *lotsfree* addiert
  - entspricht Vorschau auf künftige große Speicheranforderungen

8.6 Ersetzungsstrategie bei Solaris (4)

- Seitenuntersuchungsrate des *pageout* Prozesses

Seiten pro Sekunde



  - je weniger freier Speicher verfügbar ist, desto höher wird die Untersuchungsrate
  - *slowscan* und *fastscan* sind einstellbar

8.6 Ersetzungsstrategie bei Solaris (5)

- Weitere Parameter
  - *maxpgio:* maximale Transferrate bei Auslagerungen (vermeidet Plattensaturierung)
  - *autoup:* Zeitdauer des regelmäßigen Auslagerns alter Seiten durch den Prozess *flushd* (Default: alle 30 sec)
- Aktivieren und Deaktivieren *(Swap in, Swap out)*
  - Auswahl wird dem Scheduler überlassen
  - Deaktivierung wird lediglich von Speicherverwaltung angestoßen

8.6 Ersetzungsstrategie bei Solaris (6)

- Typische Werte
  - *minfree:* 1/64 des Hauptspeichers (Solaris 2.2), 25 Seiten (Solaris 2.4)
  - *desfree:* 1/32 des Hauptspeichers (Solaris 2.2), 50 Seiten (Solaris 2.4)
  - *lotsfree:* 1/16 des Hauptspeichers (Solaris 2.2), 128 Seiten (Solaris 2.4)
  - *deficit:* 0 bis *lotsfree*
  - *fastscan:* min(1/4 Hauptspeicher, 64 MByte) pro Sekunde (Solaris 2.4)
  - *slowscan:* 800 kBytes pro Sekunde (Solaris 2.4)
  - *handspread:* wie *fastscan* (Solaris 2.4)

8.7 Ersetzungsstrategie bei Windows 2000

- Freiliste von freien Kacheln
- Arbeitsmenge pro Prozess
  - zunächst vorbestimmt
  - Anpassung der Arbeitsmenge durch Working-Set-Manager
  - Arbeitszyklus des Working-Set-Managers wird durch Speicherknappheit beschleunigt
  - Auslagerungsstrategie nach WSClock (nur Monoprozessorvariante) unter Berücksichtigung von Prozessklassen
  - prozessspezifische Konfiguration der Arbeitsmenge durch Anwender

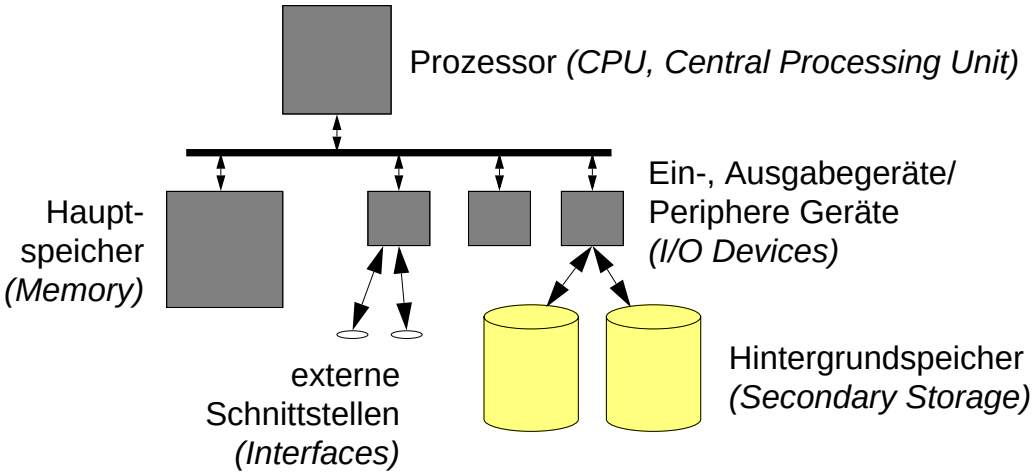
9 Zusammenfassung

- Freispeicherverwaltung
  - Speicherrepräsentation, Zuteilungsverfahren
- Mehrprogrammbetrieb
  - Relokation, Ein- und Auslagerung
  - Segmentierung
  - Seitenadressierung, Seitenadressierung und Segmentierung, TLB
  - gemeinsamer Speicher
- Virtueller Speicher
  - Demand paging
  - Seitenersetzungsstrategien: FIFO, B_0, LRU, 2nd chance (Clock)
- Seitenflattern
  - Super-Zustände, Arbeitsmengenmodell

F Implementierung von Dateien

F Implementierung von Dateien

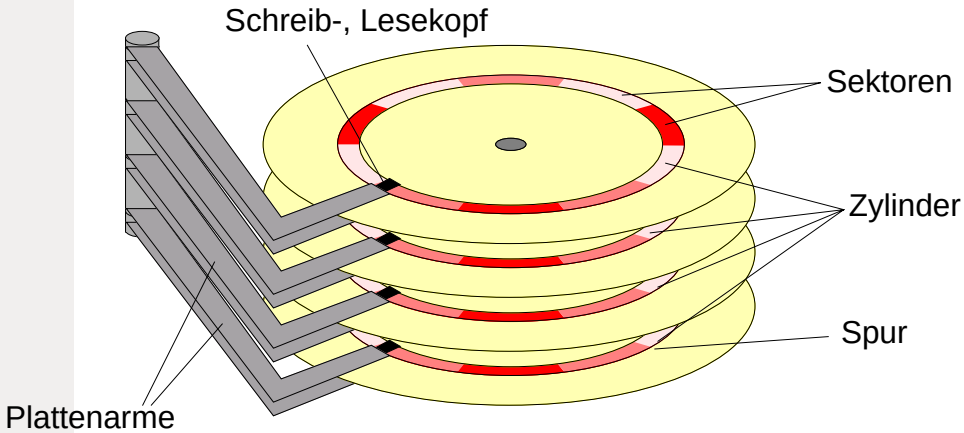
- Einordnung



1 Medien

1.1 Festplatten

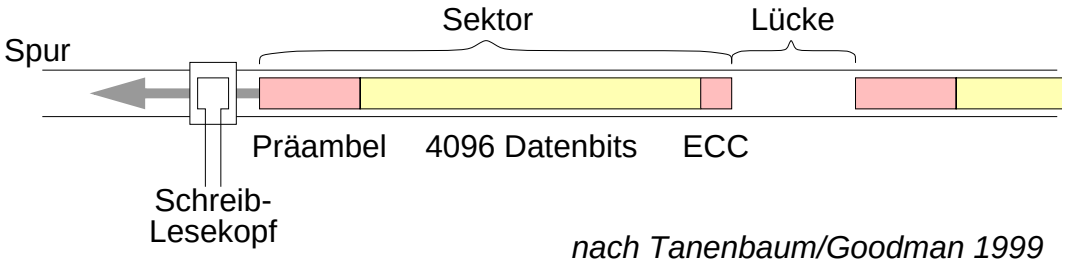
- Häufigstes Medium zum Speichern von Dateien
  - Aufbau einer Festplatte



  - Kopf schwebt auf Luftpolster

1.1 Festplatten (2)

- Sektoraufbau



nach Tanenbaum/Goodman 1999

  - Breite der Spur: 5–10 µm
  - Spuren pro Zentimeter: 800–2000
  - Breite einzelner Bits: 0,1–0,2 µm

- Zonen
  - Mehrere Zylinder (10–30) bilden eine Zone mit gleicher Sektorenanzahl (bessere Plattenausnutzung)

1.1 Festplatten (3)

- Datenblätter zweier Beispielplatten

Plattentyp		Seagate Medialist	Seagate Cheetah
Kapazität		10,2 GB	36,4 GB
Platten/Köpfe		3/6	12/24
Zylinderzahl		CHS 16383/16/83	9772
Cache		512 kB	4 MB
Positionier-zeiten	Spur zu Spur		0,6/0,9 ms
	mittlere	9,5 ms	5,7/6,5 ms
	maximale		12/13 ms
Transferrate		8,5 MB/s	18,3–28 MB/s
Rotationsgeschw.		5.400 U/min	10.000 U/min
eine Plattenumdrehung		11 ms	6 ms
Stromaufnahme		4,5 W	14 W

1.1 Festplatten (4)

- Zugriffsmerkmale
  - blockorientierter und wahlfreier Zugriff
  - Blockgröße zwischen 32 und 4096 Bytes (typisch 512 Bytes)
  - Zugriff erfordert Positionierung des Schwenkarms auf den richtigen Zylinder und Warten auf den entsprechenden Sektor
- Blöcke sind üblicherweise numeriert
  - getrennte Numerierung: Zylindernummer, Sektornummer
  - kombinierte Numerierung: durchgehende Nummern über alle Sektoren (Reihenfolge: aufsteigend innerhalb eines Zylinders, dann folgender Zylinder, etc.)

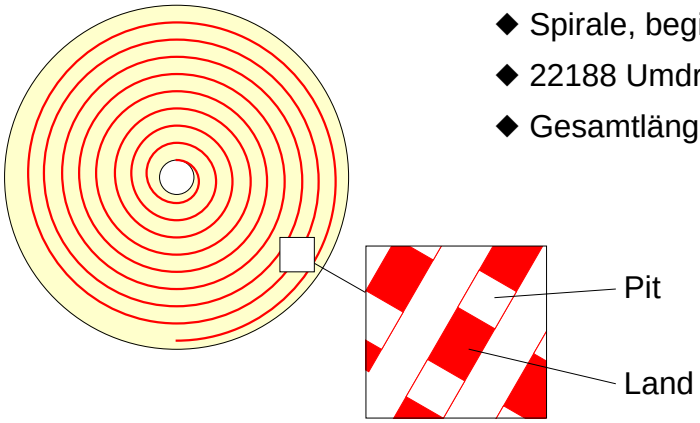
1.2 Disketten

- Ähnlicher Aufbau wie Festplatten
  - maximal zwei Schreib-, Leseköpfe (oben, unten)
  - Kopf berührt Diskettenoberfläche
- Typische Daten

Diskettentyp	3,5" HD
Kapazität	1,44 MB
Köpfe	2
Spuren	80
Sektoren pro Spur	18
Transferrate	62,5 kB/s
Rotationsgeschw.	300 U/min
eine Umdrehung	200 ms

1.3 CD-ROM

- Aufbau einer CD



- Spirale, beginnend im Inneren
- 22188 Umdrehungen (600 pro mm)
- Gesamtlänge 5,6 km

- **Pit**: Vertiefung, die von einem Laser abgetastet werden kann

1.3 CD-ROM (2)

- Kodierung
  - **Symbol**: ein Byte wird mit 14 Bits kodiert
    (kann bereits bis zu zwei Bitfehler korrigieren)
  - **Frame**: 42 Symbole werden zusammengefasst
    (192 Datenbits, 396 Fehlerkorrekturbits)
  - **Sektor**: 98 Frames werden zusammengefasst
    (16 Bytes Präambel, 2048 Datenbytes, 288 Bytes Fehlerkorrektur)
  - *Effizienz*: 7203 Bytes transportieren 2048 Nutzbytes
- Transferrate
  - Single-Speed-Laufwerk:
    75 Sektoren pro Sekunde (153.600 Bytes pro Sekunde)
  - 40-fach-Laufwerk:
    3000 Sektoren pro Sekunde (6.144.000 Bytes pro Sekunde)

1.3 CD-ROM (3)

- Kapazität
  - ca. 650 MB
- Varianten
  - **CD-R** (Recordable): einmal beschreibbar
  - **CD-RW** (Rewritable): mehrfach beschreibbar
- DVD (Digital Versatile Disk)
  - kleinere Pits, engere Spirale, andere Laserlichfarbe
  - einseitig oder zweiseitig beschrieben
  - ein- oder zweischichtig beschrieben
  - Kapazität: 4,7 bis 17 GB

2 Speicherung von Dateien

- Dateien benötigen oft mehr als einen Block auf der Festplatte
  - Welche Blöcke werden für die Speicherung einer Datei verwendet?

2.1 Kontinuierliche Speicherung

- Datei wird in Blöcken mit aufsteigenden Blocknummern gespeichert
  - Nummer des ersten Blocks und Anzahl der Folgeblöcke muss gespeichert werden
- ★ Vorteile
  - Zugriff auf alle Blöcke mit minimaler Positionierzeit des Schwenkarms
  - Schneller direkter Zugriff auf bestimmter Dateiposition
  - Einsatz z.B. bei Systemen mit Echtzeitanforderungen

2.1 Kontinuierliche Speicherung (2)

▲ Probleme

- ◆ Finden des freien Platzes auf der Festplatte (Menge aufeinanderfolgender und freier Plattenblöcke)
- ◆ Fragmentierungsproblem (Verschnitt: nicht nutzbare Plattenblöcke; siehe auch Speicherverwaltung)
- ◆ Größe bei neuen Dateien oft nicht im Voraus bekannt
- ◆ Erweitern ist problematisch
  - Umkopieren, falls kein freier angrenzender Block mehr verfügbar

2.1 Kontinuierliche Speicherung (3)

■ Variation

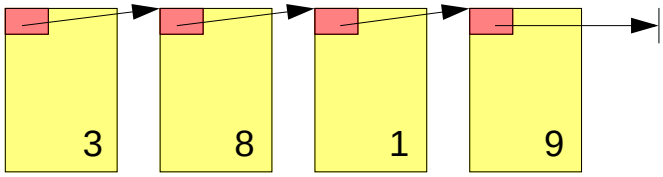
- ◆ Unterteilen einer Datei in Folgen von Blocks *(Chunks, Extents)*
- ◆ Blockfolgen werden kontinuierlich gespeichert
- ◆ Pro Datei muss erster Block und Länge jedes einzelnen Chunks gespeichert werden

▲ Problem

- ◆ Verschnitt innerhalb einer Folge (siehe auch Speicherverwaltung: interner Verschnitt bei Seitenadressierung)

2.2 Verkettete Speicherung

- Blöcke einer Datei sind verkettet



  - z.B. Commodore Systeme (CBM 64 etc.)
    - Blockgröße 256 Bytes
    - die ersten zwei Bytes bezeichnen Spur- und Sektornummer des nächsten Blocks
    - wenn Spurnummer gleich Null: letzter Block
    - 254 Bytes Nutzdaten

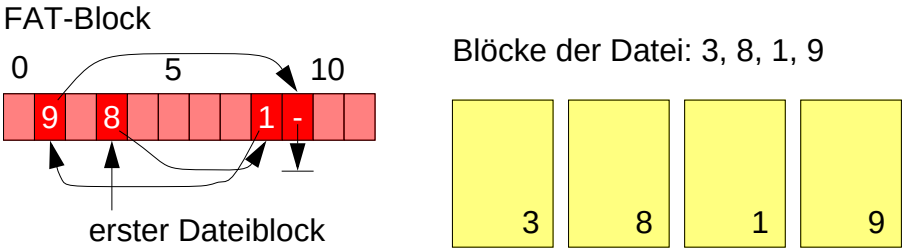
- File kann wachsen und verlängert werden

2.2 Verkettete Speicherung (2)

- Probleme
  - Speicher für Verzeigerung geht von den Nutzdaten im Block ab (ungünstig im Zusammenhang mit Paging: Seite würde immer aus Teilen von zwei Plattenblöcken bestehen)
  - Fehleranfälligkeit: Datei ist nicht restaurierbar, falls einmal Verzeigerung fehlerhaft
  - schlechter direkter Zugriff auf bestimmte Dateiposition
  - häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

2.2 Verkettete Speicherung (3)

- Verkettung wird in speziellen Plattenblocks gespeichert
  - FAT-Ansatz *(FAT: File Allocation Table)*, z.B. MS-DOS, Windows 95



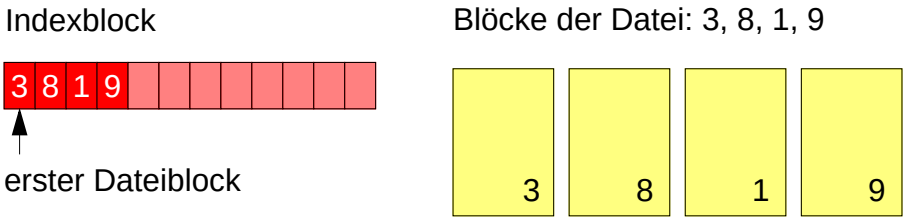
- ★ Vorteile
  - kompletter Inhalt des Datenblocks ist nutzbar (günstig bei Paging)
  - mehrfache Speicherung der FAT möglich: Einschränkung der Fehleranfälligkeit

2.2 Verkettete Speicherung (4)

- ▲ Probleme
  - mindestens ein zusätzlicher Block muss geladen werden (Caching der FAT zur Effizienzsteigerung nötig)
  - FAT enthält Verkettungen für alle Dateien: das Laden der FAT-Blöcke lädt auch nicht benötigte Informationen
  - aufwändige Suche nach dem zugehörigen Datenblock bei bekannter Position in der Datei
  - häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

2.3 Indiziertes Speichern

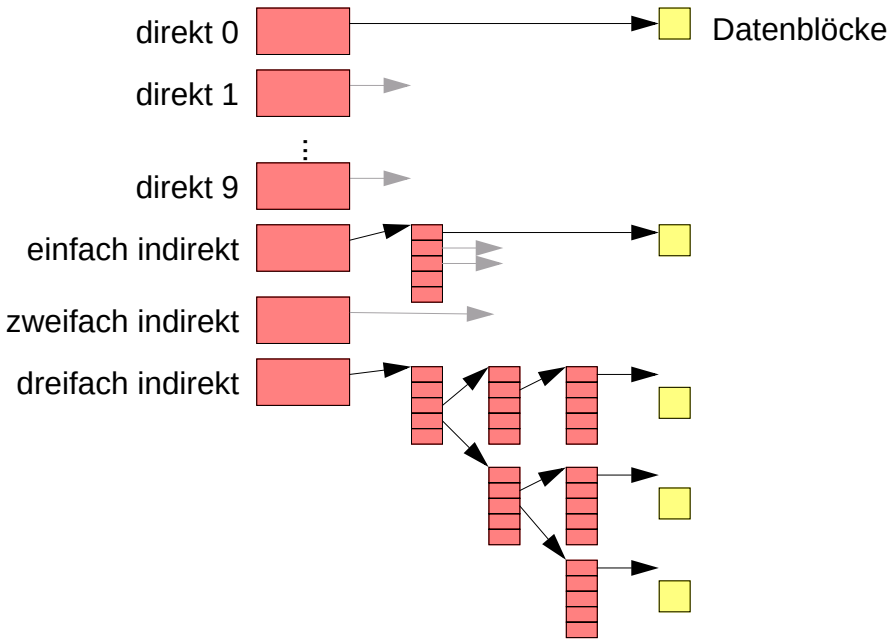
- Spezieller Plattenblock enthält Blocknummern der Datenblocks einer Datei



- Problem
  - feste Anzahl von Blöcken im Indexblock
    - Verschnitt bei kleinen Dateien
    - Erweiterung nötig für große Dateien

2.3 Indiziertes Speichern (2)

- Beispiel UNIX Inode



2.3 Indiziertes Speichern (3)

★ Einsatz von mehreren Stufen der Indizierung

- ◆ Inode benötigt sowieso einen Block auf der Platte (Verschnitt unproblematisch bei kleinen Dateien)
- ◆ durch mehrere Stufen der Indizierung auch große Dateien adressierbar

▲ Nachteil

- ◆ mehrere Blöcke müssen geladen werden (nur bei langen Dateien)

2.4 Baumsequentielle Speicherung

■ Satzorientierte Dateien

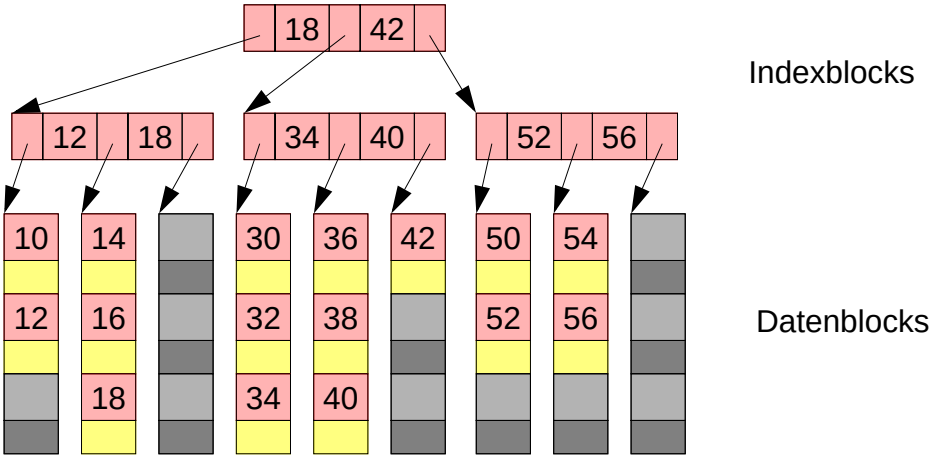
- ◆ Schlüssel + Datensatz
- ◆ effizientes Auffinden des Datensatz mit einem bekannten Schlüssel
- ◆ Schlüsselmenge spärlich besetzt
- ◆ häufiges Einfügungen und Löschen von Datensätzen

■ Einsatz von B-Bäumen zur Satzspeicherung

- ◆ innerhalb von Datenbanksystemen
- ◆ als Implementierung spezieller Dateitypen kommerzieller Betriebssysteme
  z.B. VSAM-Dateien in MVS *(Virtual Storage Access Method)*
  z.B. NTFS Katalogimplementierung

2.4 Baumsequentielle Speicherung (2)

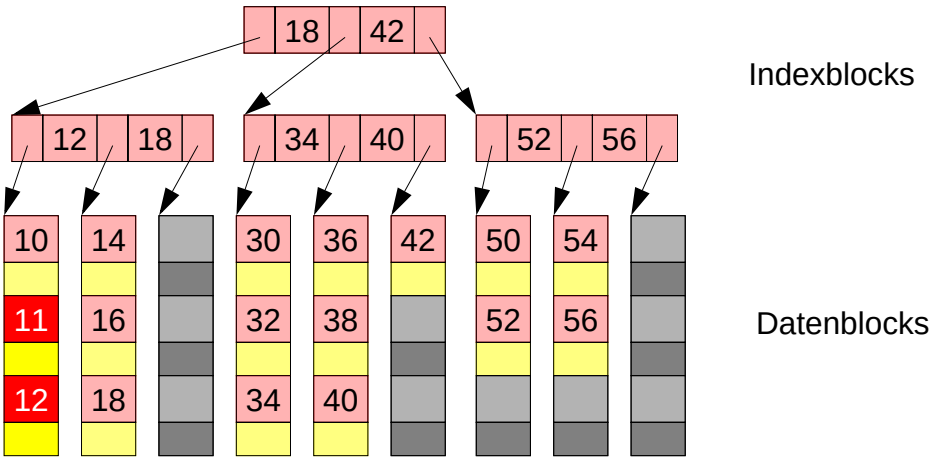
- Beispiel eines B*-Baums: Schlüssel sind Integer-Zahlen



- Blöcke enthalten Verweis auf nächste Ebene und den höchsten Schlüssel der nächsten Ebene
- Blocks der untersten Ebene enthalten Schlüssel und Sätze

2.4 Baumsequentielle Speicherung (3)

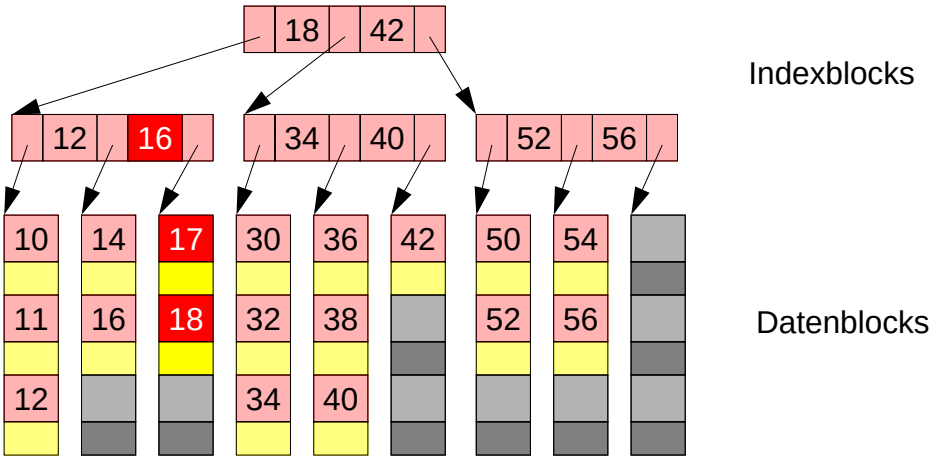
- Einfügen des Satzes mit Schlüssel „11“



- Satz mit Schlüssel „12“ wird verschoben
- Satz mit Schlüssel „11“ in freien Platz eingefügt

2.4 Baumsequentielle Speicherung (4)

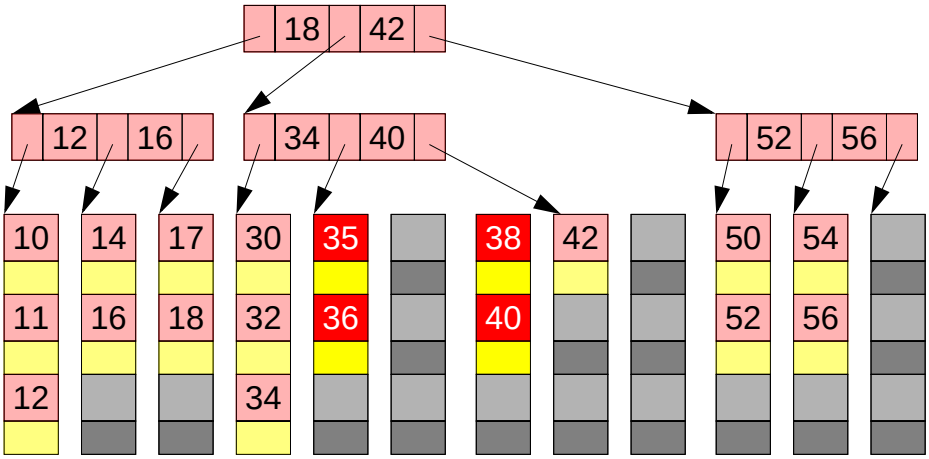
- Einfügen des Satzes mit Schlüssel „17“



- ◆ Satz mit Schlüssel „18“ wird verschoben (Indexblock wird angepasst)
- ◆ Satz mit Schlüssel „17“ in freien Platz eingefügt

2.4 Baumsequentielle Speicherung (5)

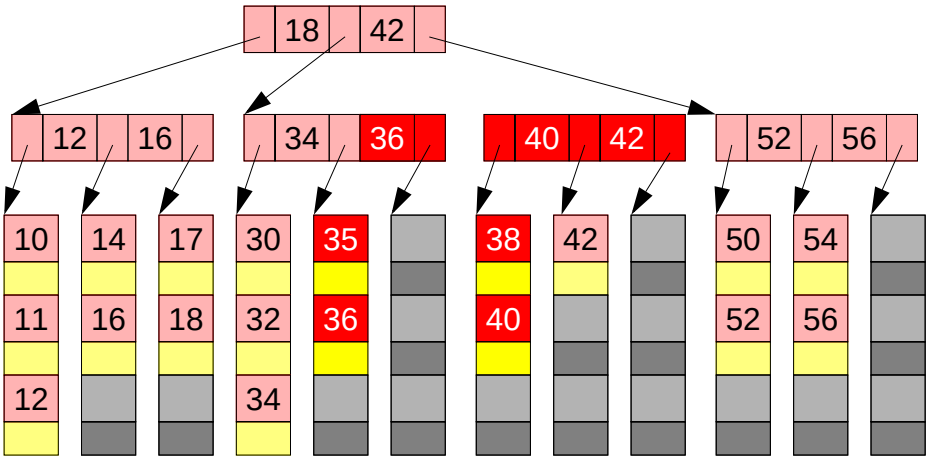
- Einfügen des Satzes mit Schlüssel „35“ (1. Schritt)



- ◆ Teilung des Blocks mit Satz „36“ und Einfügen des Satzes „35“
- ◆ Anfordern zweier weiterer, leerer Datenblöcke

2.4 Baumsequentielle Speicherung (6)

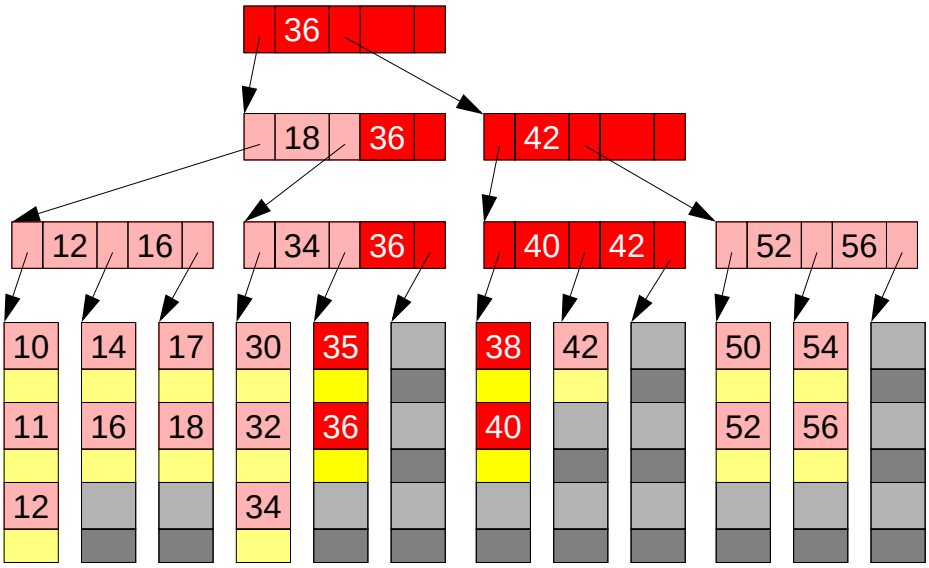
- Einfügen des Satzes mit Schlüssel „35“ (2. Schritt)



◆ Teilung bzw. Erzeugung eines neuen Indexblocks und dessen Verzeigerung

2.4 Baumsequentielle Speicherung (7)

- Einfügen des Satzes mit Schlüssel „35“ (3. Schritt)



◆ Spaltung des alten Wurzelknotens, Erzeugen eines neuer neuen Wurzel

2.4 Baumsequentielle Speicherung (8)

★ Effizientes Finden von Sätzen
- ◆ Baum ist sehr niedrig im Vergleich zur Menge der Sätze
  - viele Schlüssel pro Indexblock vorhanden (je nach Schlüssellänge)

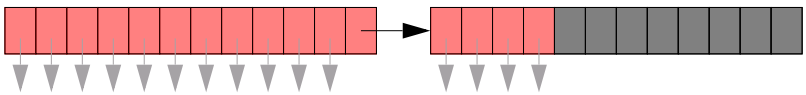
★ Gutes Verhalten im Zusammenhang mit Paging
- ◆ jeder Block entspricht einer Seite
- ◆ Demand paging sorgt für das automatische Anhäufen der oberen Indexblocks im Hauptspeicher
  - schneller Zugriff auf die Indexstrukturen

★ Erlaubt nebenläufige Operationen durch geeignetes Sperren von Indexblöcken

■ Löschen erfolgt ähnlich wie Einfügen
- ◆ Verschmelzen von schlecht belegten Datenblöcken nötig

3 Freispeicherverwaltung

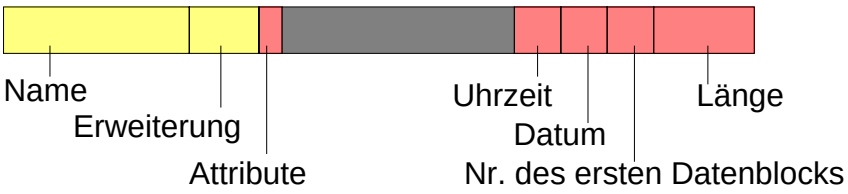
■ Prinzipiell ähnlich wie Verwaltung von freiem Hauptspeicher
- ◆ Bitvektoren zeigen für jeden Block Belegung an
- ◆ verkettete Listen repräsentieren freie Blöcke
  - Verkettung kann in den freien Blöcken vorgenommen werden
  - Optimierung: aufeinanderfolgende Blöcke werden nicht einzeln aufgenommen, sondern als Stück verwaltet
  - Optimierung: ein freier Block enthält viele Blocknummern weiterer freier Blöcke und evtl. die Blocknummer eines weiteren Blocks mit den Nummern freier Blöcke



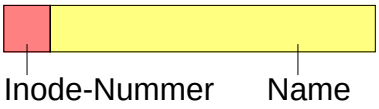
4 Implementierung von Katalogen

4.1 Kataloge als Liste

- Einträge gleicher Länge werden hintereinander in eine Liste gespeichert
  - z.B. *FAT File systems*



  - für *VFAT* werden mehrere Einträge zusammen verwendet, um den langen Namen aufzunehmen
  - z.B. *UNIX System V.3*



4.1 Kataloge als Liste (2)

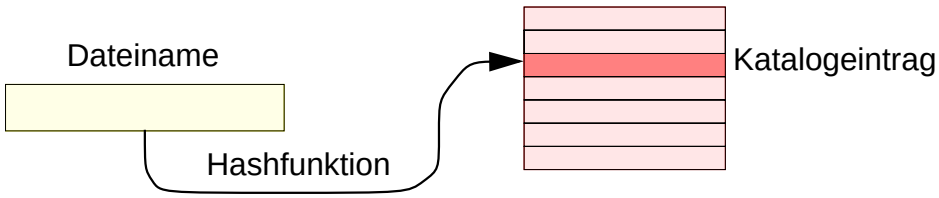
- Problem
  - Lineare Suche durch die Liste nach bestimmtem Eintrag
  - Sortierte Liste: binäre Suche, aber Sortieraufwand

4.2 Einsatz von Hashfunktionen

- Hashing
  - Spärlich besetzter Schlüsselraum wird auf einen anderen, meist dichter besetzten Schlüsselraum abgebildet
  - Beispiel: Menge der möglichen Dateinamen wird nach [0 – N-1] abgebildet (N = Länge der Katalogliste)

4.2 Einsatz von Hashfunktionen (2)

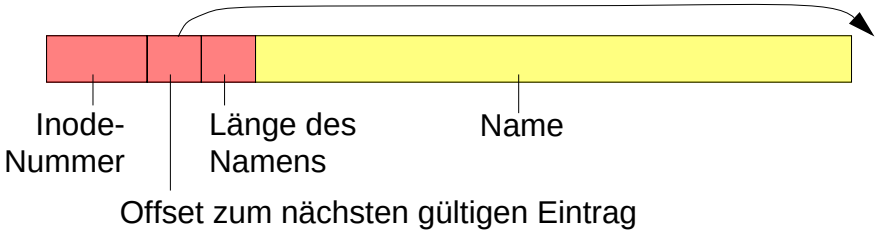
- Hashfunktion
  - Funktion bildet Dateinamen auf einen Index in die Katalogliste ab schnellerer Zugriff auf den Eintrag möglich (kein lineares Suchen)
  - (einfaches aber schlechtes) Beispiel: $(\sum \text{Zeichen}) \bmod N$



- Probleme
  - Kollisionen (mehrere Dateinamen werden auf gleichen Eintrag abgebildet)
  - Anpassung der Listengröße, wenn Liste voll

4.3 Variabel lange Listenelemente

- Beispiel *BSD 4.2, System V.4*, u.a.

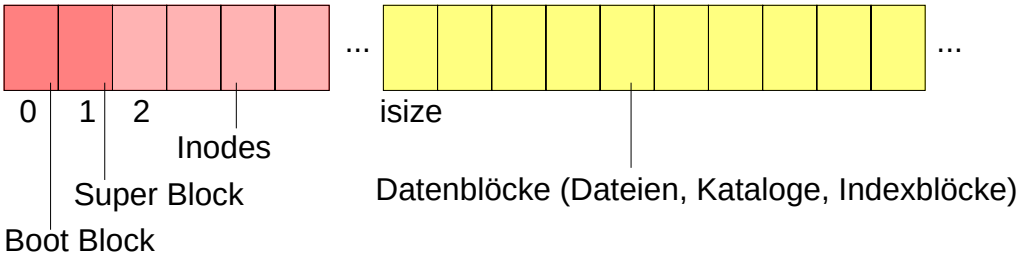


- Probleme
  - Verwaltung von freien Einträgen in der Liste
  - Speicherverschnitt (Kompaktifizieren, etc.)

5 Beispiel: UNIX File Systems

5.1 System V File System

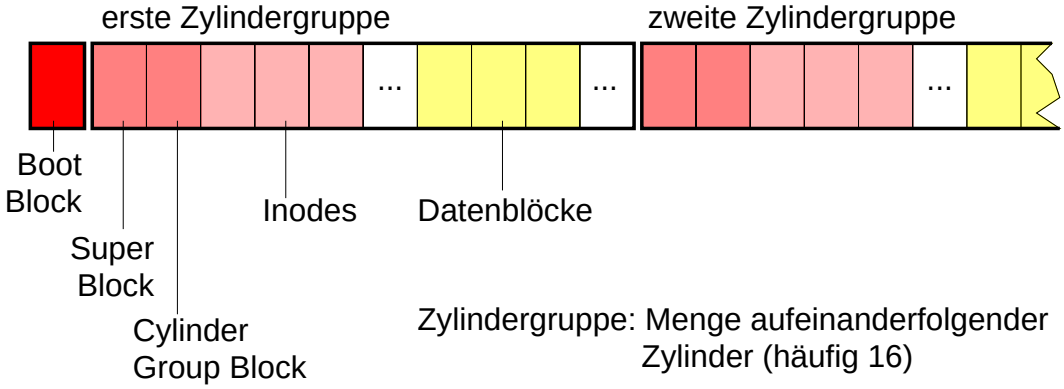
- Blockorganisation



  - Boot Block enthält Informationen zum Laden eines initialen Programms
  - Super Block enthält Verwaltungsinformation für ein Dateisystem
    - Anzahl der Blöcke, Anzahl der Inodes
    - Anzahl und Liste freier Blöcke und freier Inodes
    - Attribute (z.B. *Modified flag*)

5.2 BSD 4.2 (Berkeley Fast File System)

- Blockorganisation

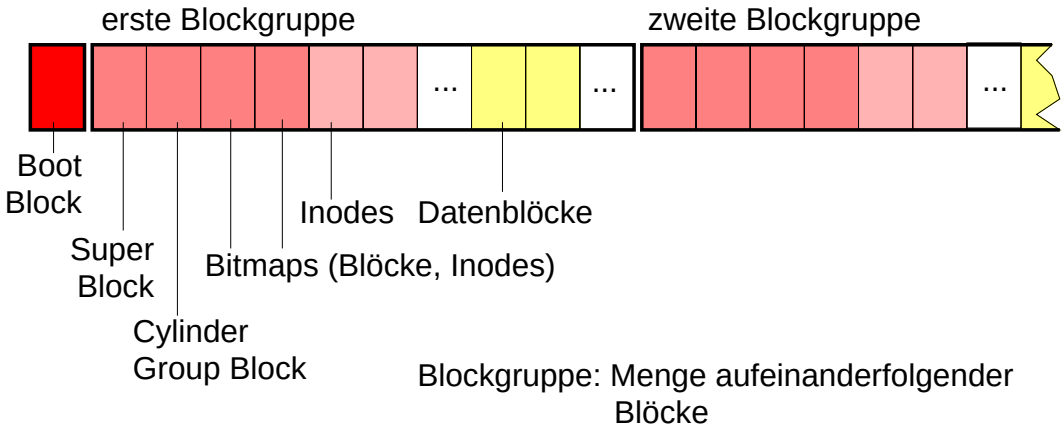


  - Kopie des Super Blocks in jeder Zylindergruppe
  - freie Inodes u. freie Datenblöcke werden im Cylinder group block gehalten
  - eine Datei wird möglichst innerhalb einer Zylindergruppe gespeichert

- ★ Vorteil: kürzere Positionierungszeiten

5.3 Linux EXT2 File System

- Blockorganisation



  - Ähnliches Layout wie BSD FFS
  - Blockgruppen unabhängig von Zylindern

5.4 Block Buffer Cache

- Pufferspeicher für alle benötigten Plattenblocks
  - Verwaltung mit Algorithmen ähnlich wie bei Paging
  - *Read ahead:* beim sequentiellen Lesen wird auch der Transfer des Folgeblocks angestoßen
  - *Lazy write:* Block wird nicht sofort auf Platte geschrieben (erlaubt Optimierung der Schreibzugriffe und blockiert den Schreiber nicht)
  - Verwaltung freier Blöcke in einer Freiliste
    - Kandidaten für Freiliste werden nach LRU Verfahren bestimmt
    - bereits freie aber noch nicht anderweitig benutzte Blöcke können reaktiviert werden *(Reclaim)*

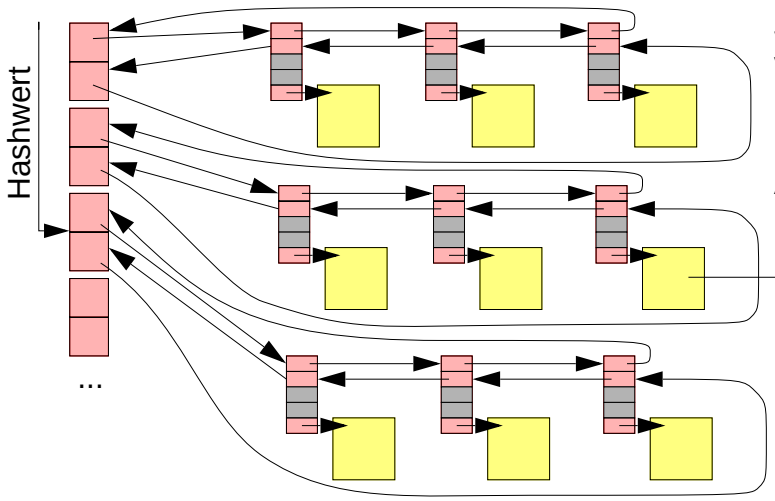
5.4 Block Buffer Cache (2)

- Schreiben erfolgt, wenn
  - Datei geschlossen wird,
  - keine freien Puffer mehr vorhanden sind,
  - regelmäßig vom System (*fsflush* Prozess, *update* Prozess),
  - beim Systemaufruf *sync(),*
  - und nach jedem Schreibaufruf im Modus *O_SYNC.*
- Adressierung
  - Adressierung eines Blocks erfolgt über ein Tupel: (Gerätenummer, Blocknummer)
  - Über die Adresse wird ein Hashwert gebildet, der eine der möglichen Pufferliste auswählt

5.4 Block Buffer Cache (3)

- Aufbau des Block buffer cache

Pufferlisten (Queues)



Jede Liste ist vorwärts und rückwärts verkettet:

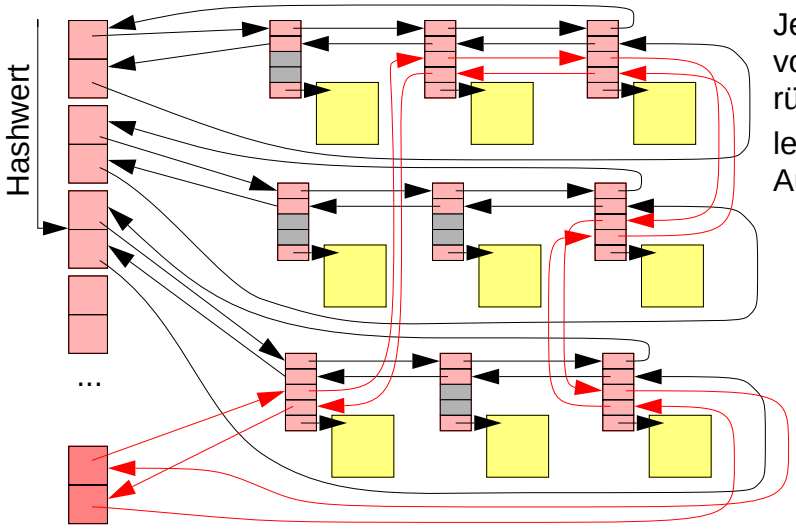
leichteres Ein- und Austragen

gepufferter Block

5.4 Block Buffer Cache (4)

- Aufbau des Block buffer cache

  Pufferlisten (Queues)



Jede Liste ist vorwärts und rückwärts verkettet

leichteres Ein- und Austragen

Freiliste

5.4 Block Buffer Cache (5)

- Block Buffer Cache teilweise obsolet durch moderne Pageing-Systeme
  - Kacheln des Hauptspeichers ersetzen den Block Buffer Cache
  - Kacheln können Seiten aus einem Adressraum und/oder Seiten aus einer Datei beherbergen
- Problem
  - Kopieren großer Dateien führt zum Auslagern noch benötigter Adressraumseiten

5.5 Systemaufrufe

- Bestimmen der Kachelgröße

```
int getpagesize( void );
```

- Abbildung von Dateien in den virtuellen Adressraum
  - ◆ Einblenden einer Datei

```
caddr_t mmap( caddr_t addr, size_t len, int prot, int flags,
              int fd, off_t off );
```

    - Einblenden an bestimmte oder beliebige Adresse
    - lesbar, schreibbar, ausführbar

  - ◆ Ausblenden einer Datei

```
int munmap( caddr_t addr, size_t len );
```


5.5 Systemaufrufe (2)

- ◆ Kontrolloperation

```
int mctl( caddr_t addr, size_t len, int func, void *arg );
```

  - zum Ausnehmen von Seiten aus dem Paging (Fixieren im Hauptspeicher)
  - zum Synchronisieren mit der Datei

6 Beispiel: Windows NT (NTFS)

- File System für Windows NT
- Datei
  - einfache, unstrukturierte Folge von Bytes
  - beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
  - dynamisch erweiterbare Dateien
  - Rechte verknüpft mit NT Benutzern und Gruppen
  - Datei kann automatisch komprimiert abgespeichert werden
  - große Dateien bis zu 8.589.934.592 Gigabytes lang
  - Hard links: mehrere Einträge derselben Datei in verschiedenen Katalogen möglich

6 Beispiel: NTFS (2)

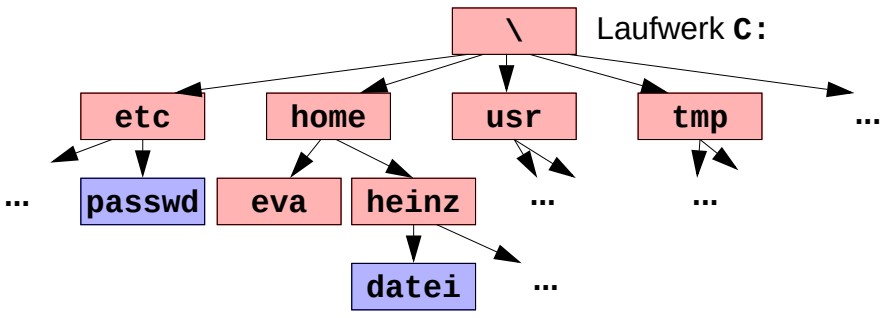
- Katalog
  - baumförmig strukturiert
    - Knoten des Baums sind Kataloge
    - Blätter des Baums sind Dateien
  - Rechte wie bei Dateien
  - alle Dateien des Katalogs automatisch komprimierbar
- Partitionen heißen Volumes
  - Volume wird (in der Regel) durch einen Laufwerksbuchstaben dargestellt z.B. **`C:`**

6.1 Rechte

- Eines der folgenden Rechte pro Benutzer oder Benutzergruppe
  - *no access:* kein Zugriff
  - *list:* Anzeige von Dateien in Katalogen
  - *read:* Inhalt von Dateien lesen und *list*
  - *add:* Hinzufügen von Dateien zu einem Katalog und *list*
  - *read&add:* wie *read* und *add*
  - *change:* Ändern von Dateiinhalten, Löschen von Dateien und *read&add*
  - *full:* Ändern von Eigentümer und Zugriffsrechten und *change*

6.2 Pfadnamen

- Baumstruktur



- Pfade
  - wie unter FAT-Filesystem
  - z.B. „**C:\home\heinz\datei**“, „**\tmp**“, „**C:..\heinz\datei**“

6.2 Pfadnamen (2)

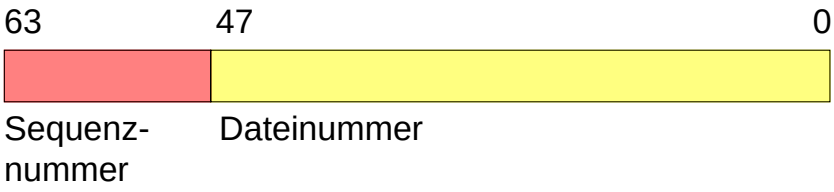
- Namenskonvention
  - 255 Zeichen inklusive Sonderzeichen (z.B. „**`Eigene Programme`**“)
  - automatischer Kompatibilitätsmodus: 8 Zeichen Name, 3 Zeichen Erweiterung, falls „langer Name“ unter MS-DOS ungültig (z.B. **`AUTOEXEC.BAT`**)
- Kataloge
  - Jeder Katalog enthält einen Verweis auf sich selbst („`.`“) und einen Verweis auf den darüberliegenden Katalog im Baum („`..`“)
  - Hard links aber keine symbolischen Namen direkt im NTFS

6.3 Dateiverwaltung

- Basiseinheit „Cluster“
  - 512 Bytes bis 4 Kilobytes (beim Formatieren festgelegt)
  - wird auf eine Menge von hintereinanderfolgenden Blöcken abgebildet
  - logische Cluster-Nummer als Adresse (LCN)
- Basiseinheit „Strom“
  - jede Datei kann mehrere (Daten-)Ströme speichern
  - einer der Ströme wird für die eigentlichen Daten verwendet
  - Dateiname, MS-DOS Dateiname, Zugriffsrechte, Attribute und Zeitstempel werden jeweils in eigenen Datenströmen gespeichert (leichte Erweiterbarkeit des Systems)

6.3 Dateiverwaltung (2)

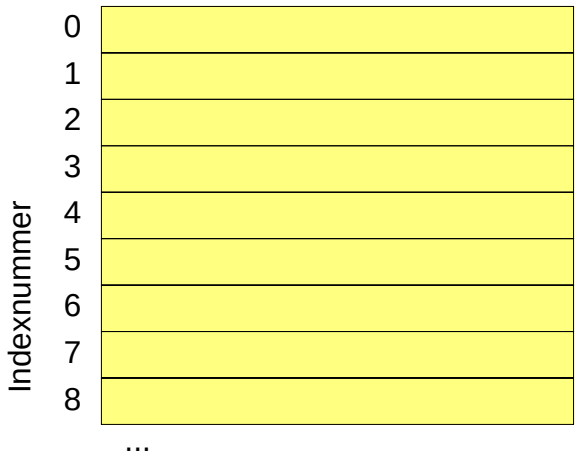
- *File-Reference*
  - Bezeichnet eindeutig eine Datei oder einen Katalog



  - Dateinummer ist Index in eine globale Tabelle *(MFT: Master File Table)*
  - Sequenznummer wird hochgezählt, für jede neue Datei mit gleicher Dateinummer

6.4 Master-File-Table

- Rückgrat des gesamten Systems
  - große Tabelle mit gleich langen Elementen
    (1KB, 2KB oder 4KB groß, je nach Clustergröße)

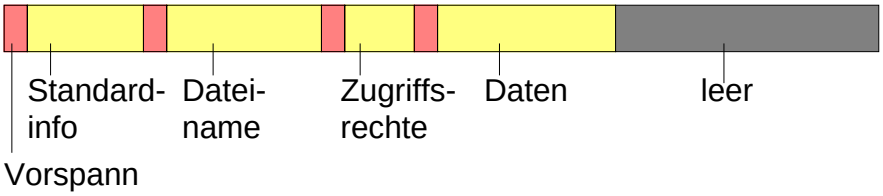


entsprechender Eintrag für eine *File-Reference* enthält Informationen über bzw. die Ströme der Datei

  - Index in die Tabelle ist Teil der *File-Reference*

6.4 Master-File-Table (2)

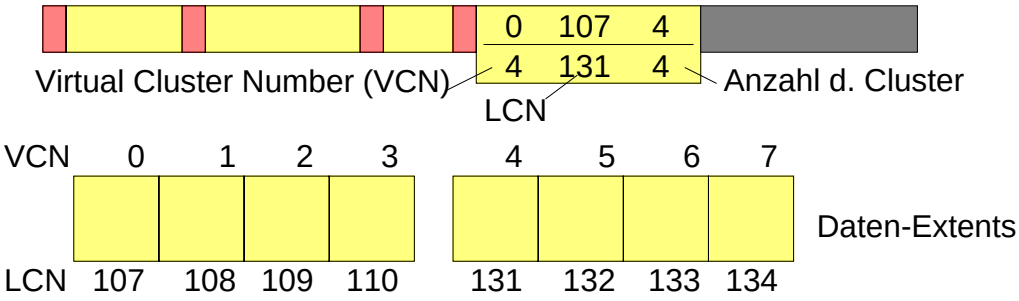
- Eintrag für eine kurze Datei



- Ströme
  - ◆ Standard Information (immer in der MFT)
    - enthält Länge, MS-DOS Attribute, Zeitstempel, Anzahl der Hard links, Sequenznummer der gültigen File-Reference
  - ◆ Dateiname (immer in der MFT)
    - kann mehrfach vorkommen (Hard links, MS-DOS Name)
  - ◆ Zugriffsrechte *(Security Descriptor)*
  - ◆ Eigentliche Daten

6.4 Master-File-Table (3)

- Eintrag für eine längere Datei



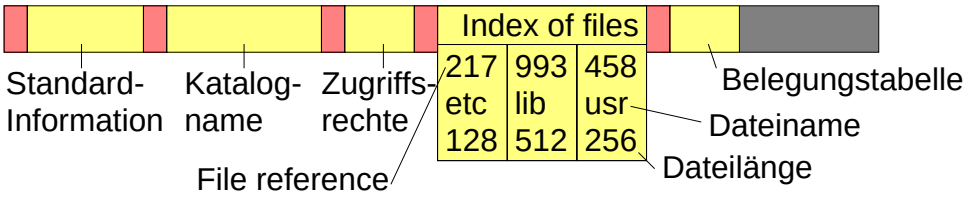
- ◆ Extents werden außerhalb der MFT in aufeinanderfolgenden Clustern gespeichert
- ◆ Lokalisierungsinformationen werden in einem eigenen Strom gespeichert

6.4 Master-File-Table (4)

- Mögliche weitere Ströme *(Attributes)*
  - ◆ Index
    - Index über einen Attributschlüssel (z.B. Dateinamen) implementiert Katalog
  - ◆ Indexbelegungstabelle
    - Belegung der Struktur eines Index
  - ◆ Attributliste (immer in der MFT)
    - wird benötigt, falls nicht alle Ströme in einen MFT Eintrag passen
    - referenzieren weitere MFT Einträge und deren Inhalt

6.4 Master File Table (3)

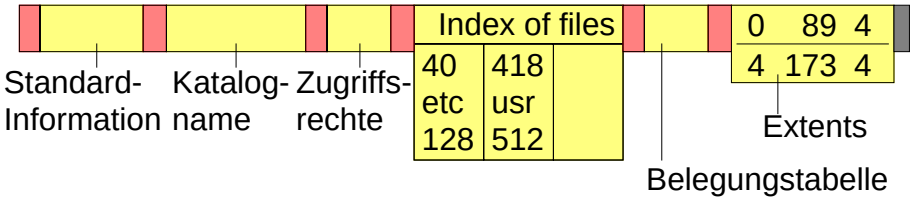
- Eintrag für einen kurzen Katalog



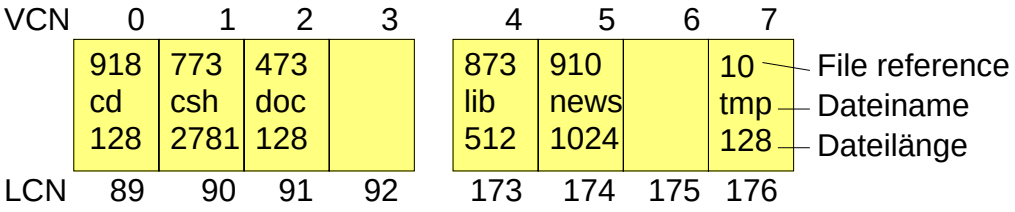
- ◆ Dateien des Katalogs werden mit File-References benannt
- ◆ Name und Länge der im Katalog enthaltenen Dateien und Kataloge werden auch im Index gespeichert
  (doppelter Aufwand beim Update; schnellerer Zugriff beim Kataloglisten)

6.4 Master File Table (4)

- Eintrag für einen längeren Katalog



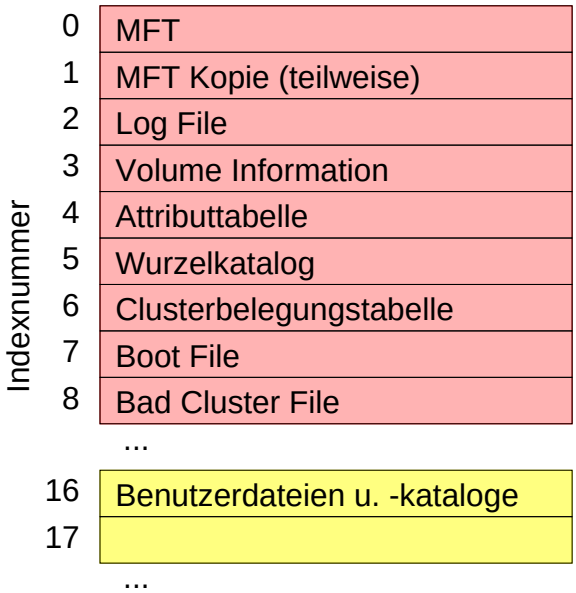
Daten-Extents



- Speicherung als B^+-Baum (sortiert, schneller Zugriff)
- in einen Cluster passen zwischen 3 und 15 Dateien (im Bild nur eine)

6.5 Metadaten

- Alle Metadaten werden in Dateien gehalten



Feste Dateien in der MFT

6.5 Metadaten (2)

- Bedeutung der Metadateien
  - MFT und MFT Kopie: MFT wird selbst als Datei gehalten
    (d.h. Cluster der MFT stehen im Eintrag 0)
    MFT Kopie enthält die ersten 16 Einträge der MFT (Fehlertoleranz)
  - Log File: enthält protokollierte Änderungen am Dateisystem
  - Volume Information: Name, Größe und ähnliche Attribute des Volumes
  - Attributtabelle: definiert mögliche Ströme in den Einträgen
  - Wurzelkatalog
  - Clusterbelegungstabelle: Bitmap für jeden Cluster des Volumes
  - Boot File: enthält initiales Programm zum Laden, sowie ersten Cluster der MFT
  - Bad Cluster File: enthält alle nicht lesbaren Cluster der Platte
    NTFS markiert automatisch alle schlechten Cluster und versucht die Daten in einen anderen Cluster zu retten

6.6 Fehlererholung

- NTFS ist ein Journaled-File-System
  - Änderungen an der MFT und an Dateien werden protokolliert.
  - Konsistenz der Daten und Metadaten kann nach einem Systemausfall durch Abgleich des Protokolls mit den Daten wieder hergestellt werden.
- Nachteile
  - etwas ineffizienter
  - nur für Volumes >400 MB geeignet

7 Dateisysteme mit Fehlererholung

- Mögliche Fehler
  - Stromausfall (dummer Benutzer schaltet einfach Rechner aus)
  - Systemabsturz
- Auswirkungen auf das Dateisystem
  - inkonsistente Metadaten
    z.B. Katalogeintrag fehlt zur Datei oder umgekehrt
    z.B. Block ist benutzt aber nicht als belegt markiert
- ★ Reparaturprogramme
  - Programme wie **`chkdsk`**, **`scandisk`** oder **`fsck`** können inkonsistente Metadaten reparieren
- ▲ Datenverluste bei Reparatur möglich
- ▲ Große Platten induzieren lange Laufzeiten der Reparaturprogramme

7.1 Journalled-File-Systems

- Zusätzlich zum Schreiben der Daten und Meta-Daten (z.B. Inodes) wird ein Protokoll der Änderungen geführt
  - Alle Änderungen treten als Teil von Transaktionen auf.
  - Beispiele für Transaktionen:
    - Erzeugen, löschen, erweitern, verkürzen von Dateien
    - Dateiattribute verändern
    - Datei umbenennen
  - Protokollieren aller Änderungen am Dateisystem zusätzlich in einer Protokolldatei *(Log File)*
  - Beim Bootvorgang wird Protokolldatei mit den aktuellen Änderungen abgeglichen und damit werden Inkonsistenzen vermieden.

7.1 Journalled-File-Systems (2)

- Protokollierung
  - Für jeden Einzelvorgang einer Transaktion wird zunächst ein Logeintrag erzeugt und
  - danach die Änderung am Dateisystem vorgenommen.
  - Dabei gilt:
    - Der Logeintrag wird immer vor der eigentlichen Änderung auf Platte geschrieben.
    - Wurde etwas auf Platte geändert, steht auch der Protokolleintrag dazu auf der Platte.

7.1 Journalled-File-Systems (3)

- Fehlererholung
  - Beim Bootvorgang wird überprüft, ob die protokollierten Änderungen vorhanden sind:
    - Transaktion kann wiederholt bzw. abgeschlossen werden *(Redo)* falls alle Logeinträge vorhanden.
    - Angefangene aber nicht beendete Transaktionen werden rückgängig gemacht *(Undo).*

7.1 Journalled-File-Systems (4)

- Beispiel: Löschen einer Datei im NTFS
  - ◆ Vorgänge der Transaktion
    - Beginn der Transaktion
    - Freigeben der Extents durch Löschen der entsprechenden Bits in der Belegungstabelle (gesetzte Bits kennzeichnen belegten Cluster)
    - Freigeben des MFT Eintrags der Datei
    - Löschen des Katalogeintrags der Datei (evtl. Freigeben eines Extents aus dem Index)
    - Ende der Transaktion
  - ◆ Alle Vorgänge werden unter der File-Reference im Log-File protokolliert, danach jeweils durchgeführt.
    - Protokolleinträge enthalten Informationen zum *Redo* und zum *Undo*

7.1 Journalled-File-Systems (5)

- ◆ Log vollständig (Ende der Transaktion wurde protokolliert und steht auf Platte):
  - *Redo* der Transaktion: alle Operationen werden wiederholt, falls nötig
- ◆ Log unvollständig (Ende der Transaktion steht nicht auf Platte):
  - *Undo* der Transaktion: in umgekehrter Reihenfolge werden alle Operation rückgängig gemacht

- Checkpoints
  - ◆ Log-File kann nicht beliebig groß werden
  - ◆ gelegentlich wird für einen konsistenten Zustand auf Platte gesorgt *(Checkpoint)* und dieser Zustand protokolliert (alle Protokolleinträge von vorher können gelöscht werden)
  - ◆ Ähnlich verfährt NTFS, wenn Ende des Log-Files erreicht wird.

7.1 Journalled-File-Systems (6)

★ Ergebnis

- ◆ eine Transaktion ist entweder vollständig durchgeführt oder gar nicht
- ◆ Benutzer kann ebenfalls Transaktionen über mehrere Dateizugriffe definieren, wenn diese ebenfalls im Log erfasst werden.
- ◆ keine inkonsistenten Metadaten möglich
- ◆ Hochfahren eines abgestürzten Systems benötigt nur den relativ kurzen Durchgang durch das Log-File.
  - Alternative **chkdsk** benötigt viel Zeit bei großen Platten

▲ Nachteile

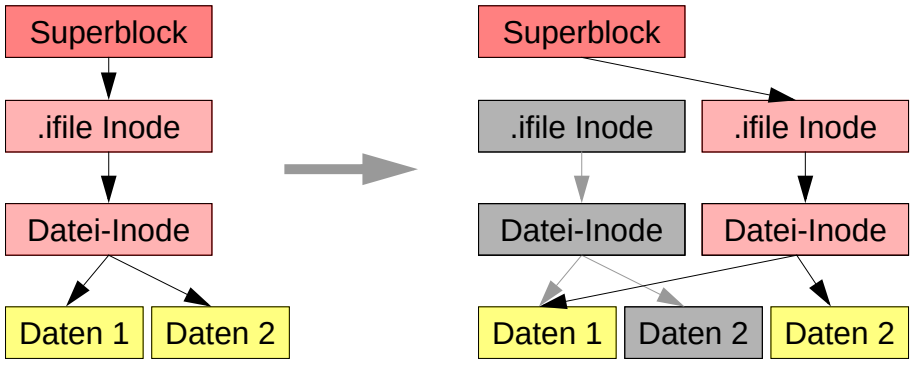
- ◆ ineffizienter, da zusätzliches Log-File geschrieben wird

■ Beispiele: NTFS, EXT3, ReiserFS

7.2 Log-Structured-File-Systems

■ Alle Änderungen im Dateisystem erfolgen auf Kopien

- ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger statischer Block (Anker im System)

7.2 Log-Structured-File-Systems (2)

★ Vorteile
- ◆ Datenkonsistenz bei Systemausfällen
  - ein atomare Änderung macht alle zusammengehörigen Änderungen sichtbar
- ◆ Schnappschüsse / Checkpoints einfach realisierbar
- ◆ Gute Schreibeffizienz
  - Alle zu schreibenden Blöcke werden kontinuierlich geschrieben

▲ Nachteile
- ◆ Gesamtperformanz geringer

■ Beispiele: LinLogFS, BSD LFS, AIX XFS

8 Limitierung der Plattennutzung

■ Mehrbenutzersysteme
- ◆ einzelnen Benutzern sollen verschieden große Kontingente zur Verfügung stehen
- ◆ gegenseitige Beeinflussung soll vermieden werden (*Disk-full* Fehlermeldung)

■ Quota-Systeme (Quantensysteme)
- ◆ Tabelle enthält maximale und augenblickliche Anzahl von Blöcken für die Dateien und Kataloge eines Benutzers
- ◆ Tabelle steht auf Platte und wird vom File-System fortgeschrieben
- ◆ Benutzer erhält Disk-full Meldung, wenn sein Quota verbraucht ist
- ◆ üblicherweise gibt es eine weiche und eine harte Grenze (weiche Grenze kann für eine bestimmte Zeit überschritten werden)

9 Fehlerhafte Plattenblöcke

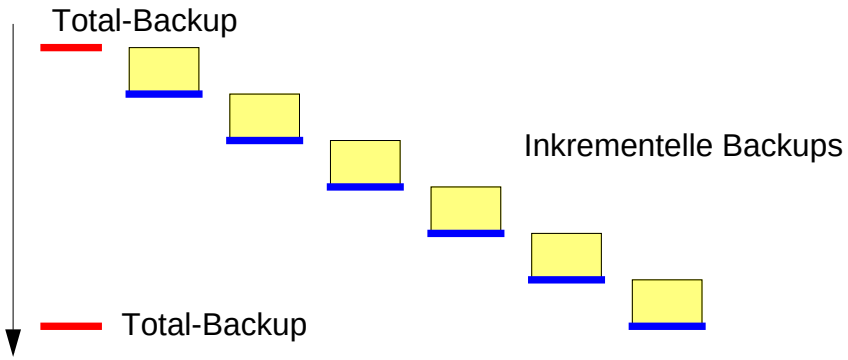
- Blöcke, die beim Lesen Fehlermeldungen erzeugen
  - z.B. Prüfsummenfehler
- Hardwarelösung
  - Platte und Plattencontroller bemerken selbst fehlerhafte Blöcke und maskieren diese aus
  - Zugriff auf den Block wird vom Controller automatisch auf einen „gesunden“ Block umgeleitet
- Softwarelösung
  - File-System bemerkt fehlerhafte Blöcke und markiert diese auch als belegt

10 Datensicherung

- Schutz vor dem Totalausfall von Platten
  - z.B. durch Head-Crash oder andere Fehler
- Sichern der Daten auf Tertiärspeicher
  - Bänder
  - WORM Speicherplatten *(Write Once Read Many)*
- Sichern großer Datenbestände
  - Total-Backups benötigen lange Zeit
  - Inkrementelle Backups sichern nur Änderungen ab einem bestimmten Zeitpunkt
  - Mischen von Total-Backups mit inkrementellen Backups

10.1 Beispiele für Backup Scheduling

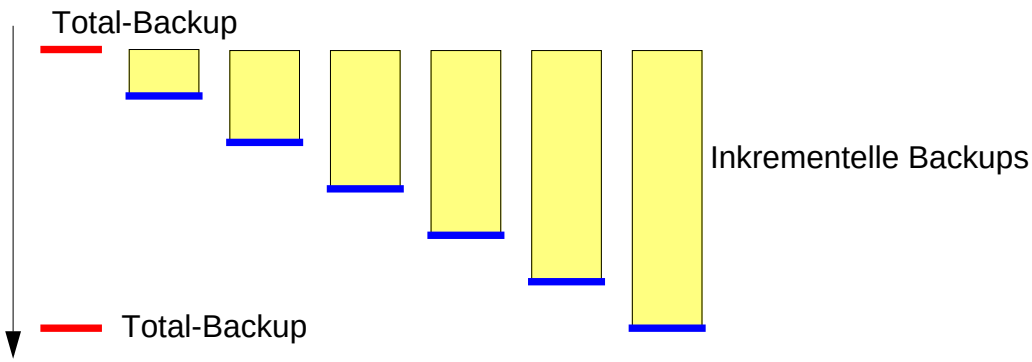
- Gestaffelte inkrementelle Backups



  - z.B. alle Woche ein Total-Backup und jeden Tag ein inkrementelles Backup zum Vortag: maximal 7 Backups müssen eingespielt werden

10.1 Beispiele für Backup Scheduling (2)

- Gestaffelte inkrementelle Backups zum letzten Total-Backup

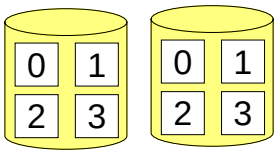


  - z.B. alle Woche ein Total-Backup und jeden Tag ein inkrementelles Backup zum letzten Total-Backup: maximal 2 Backups müssen eingespielt werden

- Hierarchie von Backup-Läufen
  - mehrstufige inkrementelle Backups zum Backup der nächst höheren Stufe
  - optimiert Archivmaterial und Restaurierungszeit

10.2 Einsatz mehrere redundanter Platten

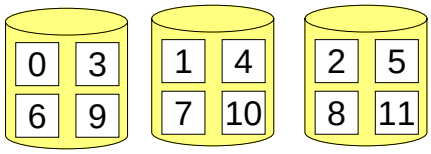
- Gespiegelte Platten (*Mirroring*; RAID 0)
  - Daten werden auf zwei Platten gleichzeitig gespeichert



  - Implementierung durch Software (File-System, Plattentreiber) oder Hardware (spez. Controller)
  - eine Platte kann ausfallen
  - schnelleres Lesen (da zwei Platten unabhängig voneinander beauftragt werden können)
- Nachteil
  - doppelter Speicherbedarf
  - wenig langsameres Schreiben durch Warten auf zwei Plattentransfers

10.2 Einsatz mehrere redundanter Platten (2)

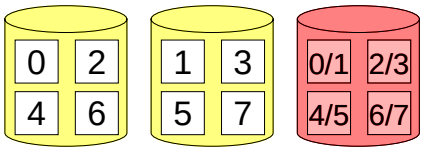
- Gestreifte Platten (*Striping*; RAID 1)
  - Daten werden über mehrere Platten gespeichert



  - Datentransfers sind nun schneller, da mehrere Platten gleichzeitig angesprochen werden können
- Nachteil
  - keinerlei Datensicherung: Ausfall einer Platte lässt Gesamtsystem ausfallen
- Verknüpfung von RAID 0 und 1 möglich (RAID 0+1)

10.2 Einsatz mehrere redundanter Platten (3)

- Paritätsplatte (RAID 4)
  - Daten werden über mehrere Platten gespeichert, eine Platte enthält Parität



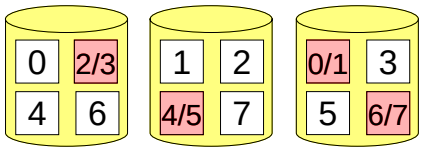
  - Paritätsblock enthält byteweise XOR-Verknüpfungen von den zugehörigen Blöcken aus den anderen Streifen
  - eine Platte kann ausfallen
  - schnelles Lesen
  - prinzipiell beliebige Plattenanzahl (ab drei)

10.2 Einsatz mehrerer redundanter Platten (4)

- Nachteil von RAID 4
  - jeder Schreibvorgang erfordert auch das Schreiben des Paritätsblocks
  - Erzeugung des Paritätsblocks durch Speichern des vorherigen Blockinhalts möglich: $P_{neu} = P_{alt} \oplus B_{alt} \oplus B_{neu}$ (P=Parity, B=Block)
  - Schreiben eines kompletten Streifens benötigt nur einmaliges Schreiben des Paritätsblocks
  - Paritätsplatte ist hoch belastet
    (meist nur sinnvoll mit SSD [Solid state disk])

10.2 Einsatz mehrere redundanter Platten (5)

- Verstreuter Paritätsblock (RAID 5)
  - Paritätsblock wird über alle Platten verstreut

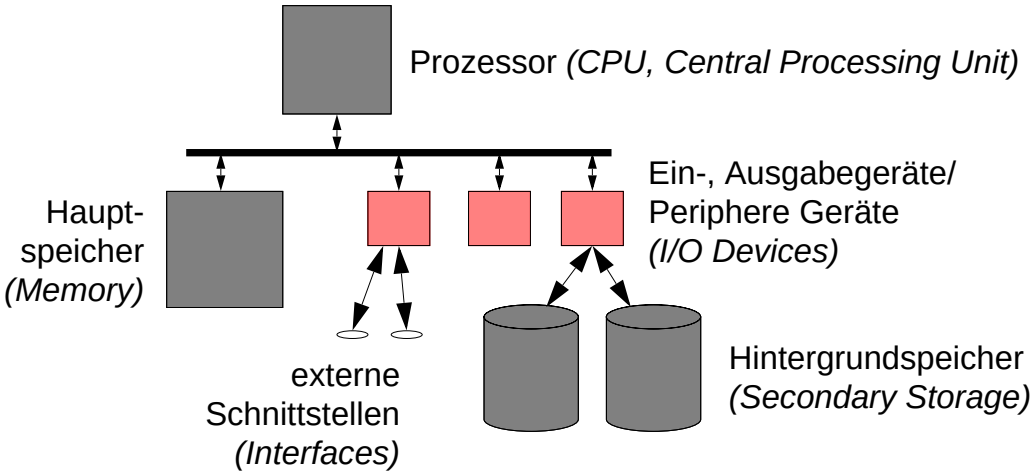


  - zusätzliche Belastung durch Schreiben des Paritätsblocks wird auf alle Platten verteilt
  - heute gängigstes Verfahren redundanter Platten
  - Vor- und Nachteile wie RAID 4

G Ein-, Ausgabe

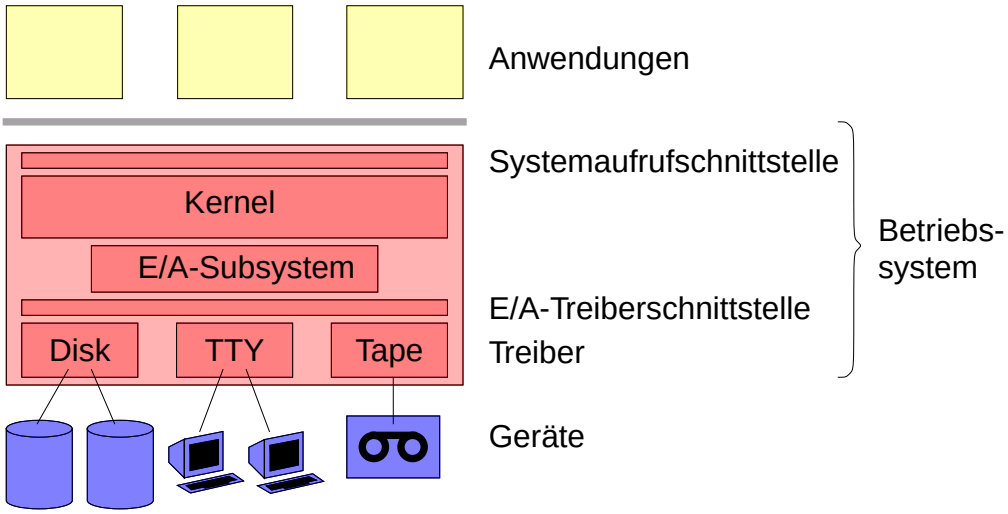
G Ein- und Ausgabe

- Einordnung



1 Gerätezugang und Treiber

- Schichtung der Systemsoftware bis zum Gerät



Nach Vahalia, 1996

1.1 Geräterepräsentation in UNIX

- Periphere Geräte werden als Spezialdateien repräsentiert
  - Geräte können wie Dateien mit Lese- und Schreiboperationen angesprochen werden
  - Öffnen der Spezialdateien schafft eine Verbindung zum Gerät, die durch einen Treiber hergestellt wird
  - direkter Durchgriff vom Anwender auf den Treiber
- Blockorientierte Spezialdateien
  - Plattenlaufwerke, Bandlaufwerke, Floppy Disks, CD-ROMs
- Zeichenorientierte Spezialdateien
  - Serielle Schnittstellen, Drucker, Audiokanäle etc.
  - blockorientierte Geräte haben meist auch eine zusätzliche zeichenorientierte Repräsentation

1.1 Geräterepräsentation in UNIX (2)

- Eindeutige Beschreibung der Geräte durch ein Tupel:
  (Gerätetyp, *Major Number, Minor Number*)
  - Gerätetyp: Block Device, Character Device
  - Major Number: Auswahlnummer für einen Treiber
  - Minor Number: Auswahl eines Gerätes innerhalb eines Treibers

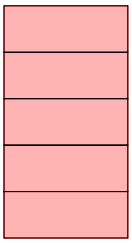
1.1 Gerätерepräsentation in UNIX (3)

- Beispiel eines Kataloglisting von **`/dev`** (Ausschnitt)

```
crw-------   1 fzhauck  108,  0 Oct 16  1996 audio
crw-------   1 fzhauck  108,128 Oct 16  1996 audioctl
crw-rw-rw-   1 root      21,  0 May  3  1996 conslog
brw-rw-rw-   1 root      36,  2 Oct 16  1996 fd0
crw-------   1 fzhauck   17,  0 Oct 16  1996 mouse
crw-rw-rw-   1 root      13,  2 Jan 13 09:09 null
crw-rw-rw-   1 root      36,  2 Jul  2  1997 rfd0
crw-r-----   1 root      32,  0 Oct 16  1996 rsd3a
crw-r-----   1 root      32,  1 Oct 16  1996 rsd3b
crw-r-----   1 root      32,  2 Oct 16  1996 rsd3c
brw-r-----   1 root      32,  0 Oct 16  1996 sd3a
brw-r-----   1 root      32,  1 Oct 16  1996 sd3b
brw-r-----   1 root      32,  2 Oct 16  1996 sd3c
crw-rw-rw-   1 root      22,  0 Sep 19 09:11 tty
crw-rw-rw-   1 root      29,  0 Oct 16  1996 ttya
crw-rw-rw-   1 root      29,  1 Oct 16  1996 ttyb
```


1.1 Gerätерepräsentation in UNIX (4)

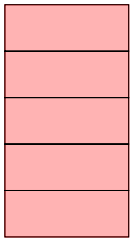
- Interne Treiberschnittstelle
  - Vektor von Funktionszeigern pro Treiber (Major Number):



`d_open`
`d_close`
`d_strategy`
`d_size`
`d_xhalt`
...

`struct bdevsw`
für blockorientierte Geräte

`struct cdevsw`
für zeichenorientierte Geräte



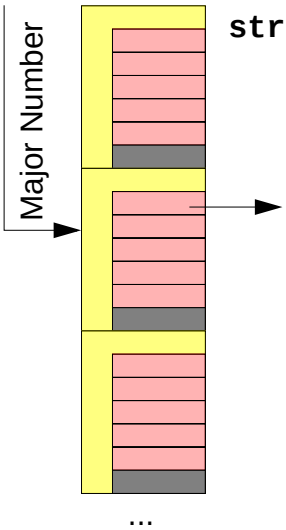
`d_open`
`d_close`
`d_write`
`d_read`
`d_ioctl`
...

1.1 Gerätерepräsentation in UNIX (5)

- Funktionen eines Block device-Treibers
  - **`d_open`**: Öffnen des Gerätes
  - **`d_close`**: Schließen des Gerätes
  - **`d_strategy`**: Abgeben von Lese- und Schreibaufträgen auf Blockbasis
  - **`d_size`**: Ermitteln der Gerätegröße (z.B. Partitions- oder Plattengröße)
  - **`d_xhalt`**: Abschalten des Gerätes
  - u.a.
- Funktionen eines Character device-Treibers
  - **`d_open, d_close`**: Öffnen und Schließen des Gerätes
  - **`d_read, d_write`**: Lesen und Schreiben von Zeichen
  - **`d_ioctl`**: generische Kontrolloperation
  - u.a.

1.1 Gerätерepräsentation in UNIX (6)

- Felder für den Aufruf von Treibern (**`bdevsw[]`** und **`cdevsw[]`**)

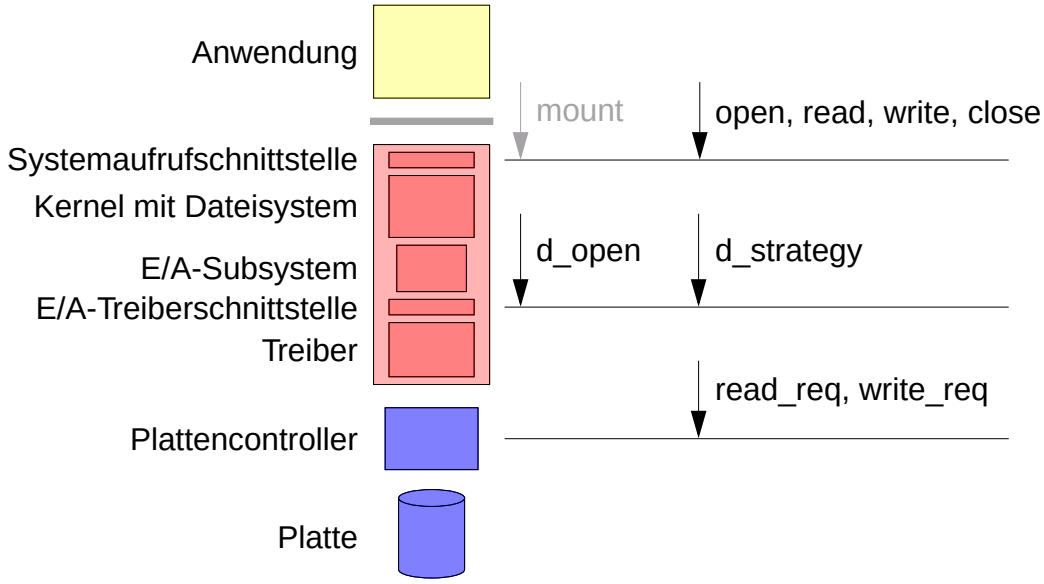


Funktion des Treibers

- *Major Number* bestimmt Element des Feldes
- Zeiger innerhalb des Feldelementes bestimmt Eintrittspunkt in die entsprechende Treiberfunktion
- *Minor Number* wird beim Aufruf als Parameter übergeben

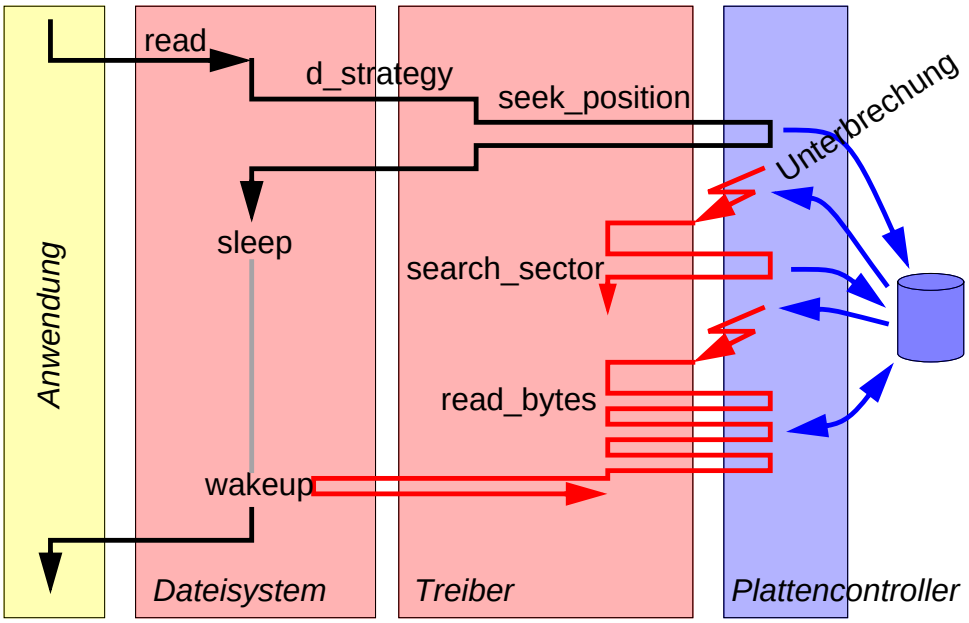
2 Plattentreiber

- Software und Hardware zwischen Anwender und Platte



2.1 Einfacher Treiber

- Ablauf eines Leseaufrufs

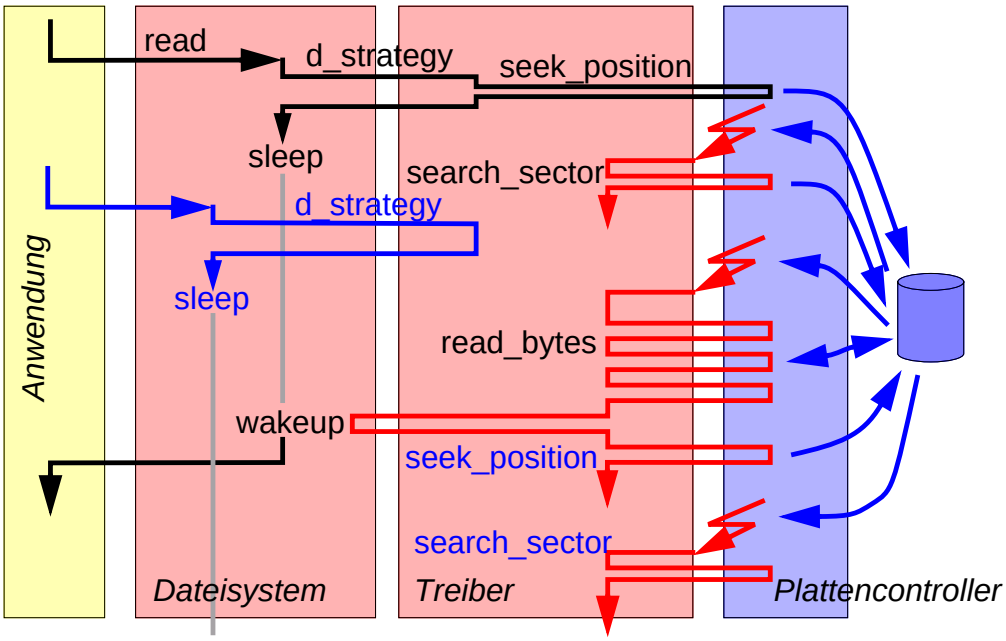


2.1 Einfacher Plattentreiber (2)

- Anwendung führt **`read()`** Systemaufruf aus.
- Dateisystem prüft, ob entsprechender Block im Speicher vorhanden.
- Falls der Block nicht vorhanden ist, wird ein Speicherplatz bereitgestellt und **`d_strategy`** im entsprechenden Treiber aufgerufen.
- Die Ausführung von **`d_strategy`** stößt Plattenpositionierung an.
- Die Anwendung blockiert sich im Kern. System kann andere Prozesse ablaufen lassen.
- Plattencontroller meldet sich bei erfolgter Positionierung durch eine Unterbrechung.
- Unterbrechungsbehandlung stößt Sektorsuche an.
- In erneuter Unterbrechung nach gefundenem Sektor werden die Daten im Pollingbetrieb eingelesen.
- Schließlich wird der Anwendungsprozess wieder aufgeweckt (in den Zustand bereit überführt).

2.1 Einfacher Plattentreiber (3)

- Ablauf mehrerer Leseaufrufe



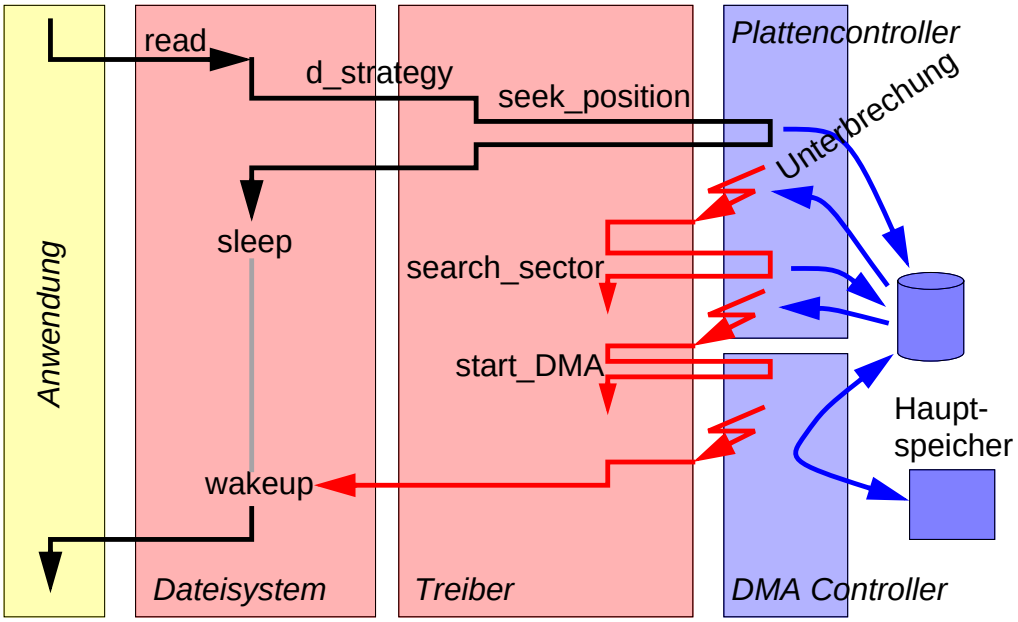
2.1 Einfacher Plattentreiber

- Unterbrechungsbehandlung ist auch für weitere Aufträge zuständig
  - Ist der Auftrag abgeschlossen, muss die Unterbrechungsbehandlung den nächsten Auftrag auswählen und aufsetzen, da der zugehörige Prozess bereits blockiert ist.
  - Die Unterbrechungen laufender Aufträge sorgen für die Abwicklung der folgenden Aufträge.

2.2 Treiber mit DMA

- DMA *(Direct Memory Access)* erlaubt Einlesen und Schreiben ohne Prozessorbeteiligung
  - DMA Controller erhält verschiedene Parameter:
    - die Hauptspeicheradresse zum Abspeichern bzw. Auslesen eines Plattenblocks
    - die Adresse des Plattencontrollers zum Abholen bzw. Abgeben der Daten
    - die Länge der zu transferierenden Daten
  - DMA Controller löst bei Fertigstellung eine Unterbrechung aus
- ★ Vorteile
  - Prozessor muss Zeichen eines Plattenblocks nicht selbst abnehmen (kein Polling sondern Interrupt)
  - Plattentransferzeit kann zum Ablauf anderer Prozesse genutzt werden

2.2 Treiber mit DMA (2)

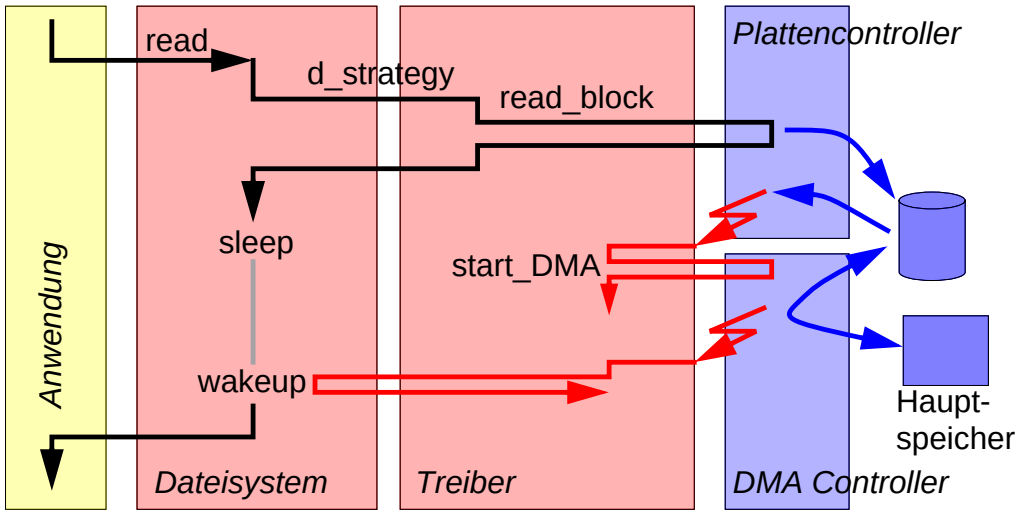


2.2 Treiber mit DMA (3)

- Große Systeme mit mehreren DMA-Kanälen und vielen Platten
  - es muss ein freier DMA-Kanal gesucht werden und evtl. auf einen freien gewartet werden bevor der Auftrag ausgeführt werden kann
  - Anforderung kann parallel zur Plattenpositionierung erfolgen
- Mainframe-Systeme
  - Steuereinheit fasst mehrere Platten zu einem Gerät zusammen
  - mehrere Steuereinheiten hängen an einem Kanal zum Hauptspeicher
  - zum Zugriff auf die eigentliche Platte muss erst die Steuereinheit und dann der Kanal belegt werden (Teilwegbelegung)
- DMA und Caching
  - heutige Prozessoren arbeiten mit Datencaches
  - DMA läuft am Cache vorbei: Betriebssystem muss vor dem Aufsetzen von DMA-Transfers Caches zurückschreiben und invalidieren

2.3 Treiber für intelligente Platte

- Intelligente Platten besitzen eigenen Prozessor für
  - das Umsortieren von Aufträgen (interne Plattenstrategie)
  - eigene Bad block-Erkennung, etc.

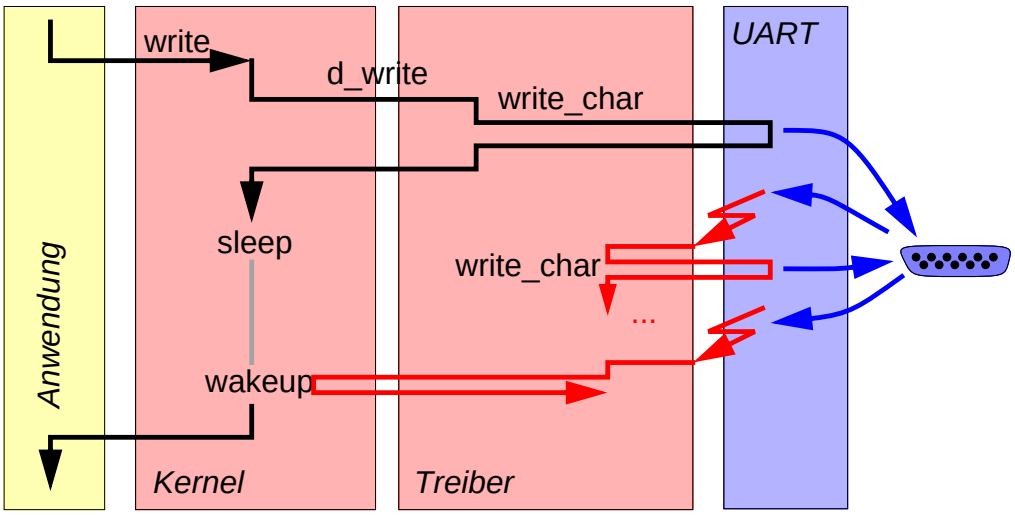


3 Treiber für serielle Schnittstellen

- Einsatz serieller Schnittstellen (z.B. RS-232)
  - Terminals
  - Drucker
  - Modems
- Datenübertragung
  - zeichenweise seriell (z.B. Startbit, Datenbits, Stopbits)
  - getaktet in bestimmter Geschwindigkeit (Bitrate, z.B. 38.400 Bit/s), im Vergleich zu Platten relativ langsam
  - Flusskontrolle (d.h. Empfänger kann Datenfluss bremsen)
  - bidirektional
- Treiber
  - zeichenorientiertes Gerät
  - vom Prinzip her ähnlich dem Plattentreiber

3.1 TTY-Treiber

- TTY-Treiber (*Teletype*, Fernschreiber) und der Ablauf eines Schreibaufrufs



  - UART = Universal Asynchronous Receiver / Transmitter

3.1 TTY-Treiber (2)

- Enger Zusammenhang zwischen Ein- und Ausgabe
  - Echofunktion (getippte Zeichen werden angezeigt)
    - eingelesene Zeichen werden gleich wieder ausgegeben
  - Flusskontrolle (bestimmtes Zeichen in der Eingabe hält Ausgabe an: ^S)
    - wird ^S eingelesen wird Ausgabe angehalten bis ^Q eingelesen wird
- Zeilenorientierte Treiber
  - Anwendung will Zeichen zeilenweise, z.B. Shell
  - Treiber blockiert Prozess bis Zeilenende erkannt
  - Treiber erlaubt das Editieren der Zeile (Backspace, etc.)
- Signale
  - bestimmte Zeichen lösen Signale an korrespondierende Prozesse aus

3.2 TTY-Treiber in UNIX

- Konfigurierbar
  - Repräsentation einer seriellen Schnittstellen als zeichenorientiertes Gerät
  - durch Aufruf von ioctl kann Treiber konfiguriert werden

```
int ioctl( int fildes, int request, /* arg */ );
```

  - Kommando zum Lesen der Konfiguration: Übergabe einer Strukturadresse

```
struct termios t;
ioctl( fd, TCGETS, &t );
```

  - Kommando zum Schreiben einer Konfiguration:

```
ioctl( fd, TCSETS, &t );
```

  - Struktur enthält Bitfelder für verschiedene Einstellungen
  - Bitmasken sind als Makros verfügbar
  - näheres: „**`man termios`**“ und „**`man ioctl`**“

3.3 Einstellung der physikalischen Parameter

- Bitrate einer seriellen Schnittstelle
  - **`B2400`** 2400 Bit/s
  - **`B4800`** 4800 Bit/s
  - **`B9600`** 9600 Bit/s
  - **`B19200`** 19200 Bit/s
  - **`B38400`** 38400 Bit/s
  - **`B57600`** 57600 Bit/s
- Zeichengröße, Parität, Stopbits
  - **`CS7`** 7 Bits
  - **`CSTOPB`** zwei Stoppbits sonst eins
  - **`PARENB`** Parität einschalten
  - **`CRTSCTS`** Hardware-basierte Flusskontrolle einschalten

3.4 Einstellung der Ein-, Ausgabeverarbeitung

- Festlegen der Zeichen mit Sonderbedeutung
  - Erase-Character: löscht letztes Zeichen (Backspace)
  - Kill-Character: löscht ganze Zeile (**^K**)
- Eingabeverarbeitung
  - **ICRNL** CR-Zeichen wird als NL-Zeichen gelesen
  - **ICANON** kanonische Eingabeverarbeitung (Zeileneditierung)
  - **IXON** erlaube Flusskontrolle mit **^Q** und **^S**
- Ausgabeverarbeitung
  - **ECHO** schaltet Echofunktion ein
  - **ECHOE** Echo von Backspace als Backspace, Leerzeichen, Backspace
  - **ONLCR** NL-Zeichen wird als CR, NL ausgegeben

3.5 Signalauslösung und Jobkontrolle

- Signalauslösung
  - **ISIG**: Schaltet Signale ein
  - **INTR**-Zeichen: sendet **SIGINT**-Signal (**^C**)
  - **QUIT**-Zeichen: sendet **SIGQUIT**-Signal (**^|**)
- Signal wird an ganze Prozessgruppe geschickt
  - alle Prozesse der Gruppe empfangen Signal
  - Beispiel: `cat /etc/passwd | grep Mueller | sort`
  - alle Prozesse erhalten **SIGINT** bei **^C**
- Prozessgruppe
  - Prozessgruppen-ID wird wie eine Prozess-ID (PID) bezeichnet
  - Prozess mit gleicher PID und Prozessgruppen-ID ist Gruppenführer
  - Shell sorgt dafür, dass im Beispiel **cat**, **grep** und **sort** in der gleichen Prozessgruppe sind (**sort** wird Gruppenführer)

3.5 Signalauslösung und Jobkontrolle (2)

- Vordergrund- und Hintergrundprozesse
  - Hintergrundprozesse erhalten keine Signale.
  - Bei Shells mit Jobkontrolle kann zwischen Vorder- und Hintergrundprozessen umgeschaltet werden.
- Sessions
  - Shell öffnet eine Session, die mehrere Prozessgruppen enthalten kann (spezieller sytemabhängiger Systemaufruf).
  - Shell wird Sessionführer.
  - Shell erzeugt Prozesse und Prozessgruppen.
  - Ein TTY wird Controlling-Terminal für alle Prozessgruppen der Session.
  - Unterbrechen der Terminalverbindung (**`SIGHUP`**) wird dem Sessionführer zugestellt.

3.5 Signalauslösung und Jobkontrolle (3)

- Vordergrundprozess
  - Eine Prozessgruppe der Session kann zur Vordergrundprozessgruppe gemacht werden.
  - **`SIGINT`** und **`SIGQUIT`** sowie die Eingabe vom Terminal werden nur der Vordergrundprozessgruppe zugestellt.
- Hintergrundprozesse
  - Alle Hintergrundprozesse bekommen keine Eingabe vom Terminal und werden gestoppt, wenn sie lesen wollen (Shell wird benachrichtigt).
- Jobkontrolle
  - Shell kann zwischen Vorder- und Hintergrundprozessgruppen umschalten
  - Benutzer kann Vordergrundprozesse stoppen und gelangt zur Shell zurück

3.5 Signalzustellung und Jobkontrolle (4)

- Beispiel: Stoppen und wiederaufnehmen eines Vordergrundprozesses

```
prompt> cc -o test.c
^Z
Suspended
prompt> jobs
[1] Suspended cc -o test.c
prompt> fg %1
```

  - Realisiert mit einem Signal namens **SIGTSTP**, das die Prozessgruppe stoppt
  - Shell bekommt dies mit über ein **waitpid()**
  - Shellkommando **fg** sendet einn Signal **SIGCONT** und die Prozesse fahren fort

3.5 Signalzustellung und Jobkontrolle (5)

- Beispiel: Stoppen eines Vordergrundprozesses, Umwandlung in einen Hintergrundprozess

```
prompt> cc -o test.c
^Z
Suspended
prompt> bg
[1] Running cc -o test.c
prompt>
```

  - Wie auf vorheriger Folie, aber:
    Shell schaltet die Prozessgruppe in den Hintergrund und wartet nicht mehr auf deren Beendigung.

3.5 Signalzustellung und Jobkontrolle (6)

- Beispiel: Starten eines Hintergrundprozesses und Umwandlung in einen Vordergrundprozess

```
prompt> cc -o test.c &
prompt> jobs
[1] Running cc -o test.c
prompt> fg %1
```

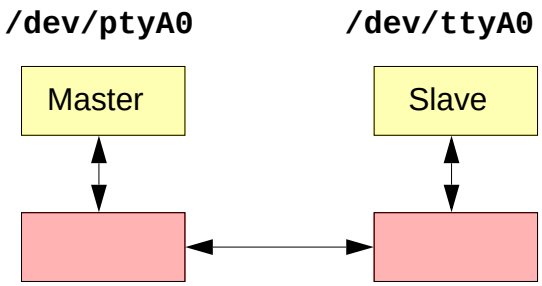
  - Shell startet eine Hintergrundprozessgruppe und nimmt Kommandos entgegen
  - **`fg`** Kommando schaltet die Hintergrundgruppe in eine Vordergrundprozessgruppe um und wartet auf deren Beendigung mit **`waitpid()`**

3.6 Pseudo-Terminals

- Pseudo-TTY-Treiber *(PTTY)*
  - keine echte serielle Schnittstelle vorhanden
  - Shell und andere Prozesse benötigen aber ein TTY für
    - Flusskontrolle,
    - Echofunktion,
    - Job-Kontrolle etc.
  - fungiert als gewohnte Schnittstelle von Anwendungsprozessen
  - Einsatz beispielsweise bei einem Fenstersystem (xterm-Programm)
    - xterm-Programm bedient die Masterseite eines PTTY
    - Shell und Anwendungsprogramme sehen xterm-Fenster wie ein TTY (Slaveseite)

3.6 Pseudo-Terminals (2)

- Master- und Slaveseite sehen wie ein normales TTY-Device aus



  - Slaveseite besitzt Modul zur Flusskontrolle, Eingabeeditierung, Signalzustellung, Flusskontrolle etc.

3.7 Warten auf mehrere Ereignisse

- Bisher: Lese- oder Schreibaufrufe blockieren
  - Was tun beim Lesen von mehreren Quellen?
- Alternative 1: nichtblockierende Ein-, Ausgabe
  - **`O_NDELAY`** beim **`open()`**
  - Pollingbetrieb: Prozess muss immer wieder **`read()`** aufrufen, bis etwas vorliegt

3.7 Warten auf mehrere Ereignisse (2)

- Alternative 2: Blockieren an mehreren Filedeskriptoren
  - Systemaufruf:

```
int select( int nfds, fd_set *readfds, fd_set *writefds,
                      fd_set *errorfds, struct timeval *timeout);
```

  - **`nfds`** legt fest, bis zu welchem Filedeskriptor **`select`** wirken soll.
  - **`XXXfds`** sind Filedeskriptoren, auf die gewartet werden soll:
    - **`readfds`** — bis etwas zum Lesen vorhanden ist
    - **`writefds`** — bis man schreiben kann
    - **`errorfds`** — bis ein Fehler aufgetreten ist
  - Timeout legt fest, wann der Aufruf spätestens deblockiert.
  - Makros zum Erzeugen der Filedeskriptormengen
  - Ergebnis: in den Filedeskriptormengen sind nur noch die Filedeskriptoren vorhanden, die zur Deblockade führten

4 Bildschirmtreiber

- Bildspeicher
  - zeichenorientiert
  - pixelorientiert
- Aufgaben des Treibers
  - Bereitstellen von Graphikprimitiven (z.B. Ausgabe von Text, Zeichnen von Rechtecken, etc.)
  - Ansprechen von Graphikprozessoren (schnelle Verschiebeoperationen, komplexe Zeichenoperationen, 3D Rendering, Textures)
  - Einblenden des Bildspeichers in Anwendungsprogramme (z.B. X11-Server)
- Bildspeicher
  - spezieller Speicher, der den Bildschirminhalt repräsentiert
  - Dual ported RAM (Videochip und Prozessor können gleichzeitig zugreifen)

5 Netzwerktreiber

- Beispiel: Ethernet
  - schneller serieller Bus mit CSMA/CD
    (Carrier sense media access / Collision detect)
    zu deutsch: es wird dann gesendet, wenn nicht gerade jemand anderes sendet; Kollisionen werden erkannt und aufgelöst
  - spezieller Netzwerkchip
    - implementiert unterstes Kommunikationsprotokoll
    - erkennt eintreffende Pakete
- Netzwerktreiber
  - wird von höheren Protokollen innerhalb des Betriebssystems angesprochen, z.B. von der IP-Schicht

5 Netzwerktreiber (2)

- Senden
  - Treiber übergibt dem Netzwerkchip eine Datenstruktur mit den notwendigen Informationen: Sendeadresse, Adresse und Länge von Datenpuffern
  - Netzwerkchip löst Unterbrechung bei erfolgtem Senden aus
- Empfangen
  - Treiber übergibt dem Netzwerkchip eine Datenstruktur mit Adressen von freien Arbeitspuffern
  - erkennt der Netzwerkchip ein Paket (für die eigene Adresse), füllt er das Paket in einen freien Puffer
  - der Puffer wird in eine Liste von empfangenen Paketen eingehängt und eine Unterbrechung ausgelöst
  - Treiber kann die empfangenen Pakete aushängen

5 Netzwerktreiber (3)

- Übertragung der Daten erfolgt durch DMA
  - evtl. direkt durch den Netzwerkchip
- Intelligente und nicht-intelligente Netzwerkhardware
  - intelligente Hardware: kann evtl. auch höhere Protokolle, Filterung etc.
  - nicht-intelligente Hardware: benötigt mehr Unterstützung durch den Treiber (Prozessor)

6 Andere Geräte

- Uhr
  - Hardwareuhren (z.B. DCF 77, GPS Empfänger)
  - Systemuhr fast immer in Software (wird mit Hardwareuhren synchronisiert)
  - UNIX: **`getitimer, setitimer`**
    - vier Intervalltimer pro Prozess: Signal **`SIGALRM`** nach Ablauf
    - Ablauf konfigurierbar:
      Realzeit, Virtuelle Zeit, Virtuelle Zeit (einschl. Systemzeit des Prozesses)
- Bandlaufwerk
  - zeichenorientiertes Gerät
  - Spuloperationen durch **`d_ioctl`** realisiert

6 Andere Geräte (2)

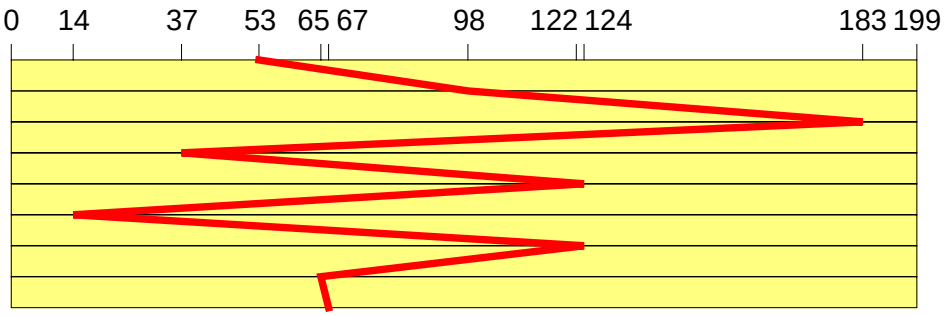
- CD-ROM
  - wird wie Platte behandelt (eigener Treiber)
  - nicht beschreibbar
  - spezielle Treiber für Audio-Tracks möglich
- Maus und Tastatur
  - meist über serielle Schnittstellen und bestimmtes Protokoll implementiert
- Floppy-Disk
  - wird im Prinzip wie Platte behandelt (eigener Treiber)
  - spezielle Dateisysteme zur Realisierung von FAT-Dateisystemen unter UNIX

7 Disk-Scheduling

- Plattentreiber hat in der Regel mehrere Aufträge in seiner Warteschlange
  - Warteschlange wird z.B. in UNIX durch Aufruf der Funktion **`d_strategy()`** gefüllt
  - eine bestimmte Ordnung der Ausführung kann Effizienz steigern
  - Zusammensetzung der Bearbeitungszeit eines Auftrags:
    - Positionierzeit: abhängig von der aktuellen Stellung des Plattenarms
    - Latenzzeit: Zeit bis der Magnetkopf den Sektor bestreicht
    - Übertragungszeit: Zeit zur Übertragung der eigentlichen Daten
- ★ Ansatzpunkt: Positionierzeit

7.1 FCFS-Scheduling

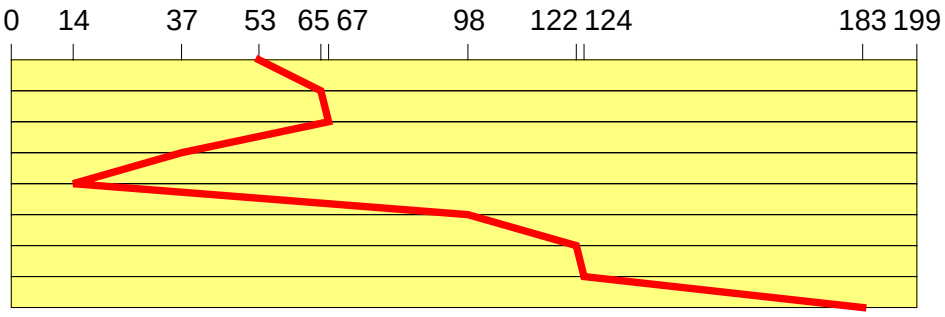
- Bearbeitung gemäß Ankunft des Auftrags
  - Referenzfolge (Folge von Zylindernummern):
    98, 183, 37, 122, 14, 124, 65, 67
  - Aktueller Zylinder: 53



  - Gesamtzahl der Spurwechsel: 640
  - Weite Bewegungen des Schwenkarms: mittlere Bearbeitungsdauer lang

7.2 SSTF-Scheduling

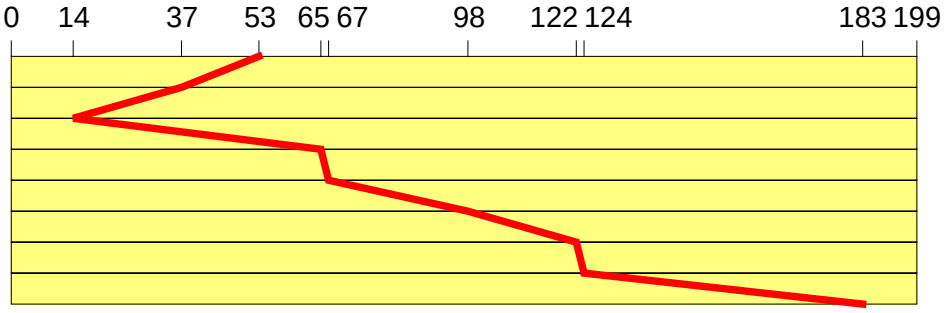
- Es wird der Auftrag mit der kürzesten Positionierzeit vorgezogen
  (Shortest Seek Time First)
  - Gleiche Referenzfolge
    (Annahme: Positionierzeit proportional zum Zylinderabstand)



  - Gesamtzahl von Spurwechseln: 236
  - ähnlich wie SJF kann auch SSTF zur Aushungerung führen
  - noch nicht optimal

7.3 SCAN-Scheduling

- Bewegung des Plattenarm in eine Richtung bis keine Aufträge mehr vorhanden sind (Fahrstuhlstrategie)
  - Gleiche Referenzfolge (Annahme: bisherige Kopfbewegung Richtung 0)

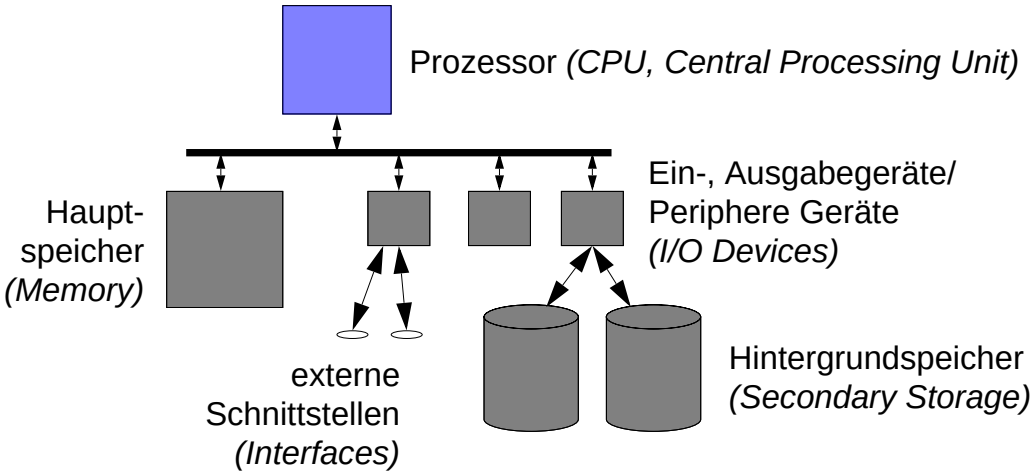


  - Gesamtzahl der Spurwechsel: 208
  - Neue Aufträge werden miterledigt ohne zusätzliche Positionierzeit und ohne mögliche Aushungerung
  - Variante C-SCAN *(Circular SCAN)*: Bewegung nur in eine Richtung

H Verklemmungen

H Verklemmungen

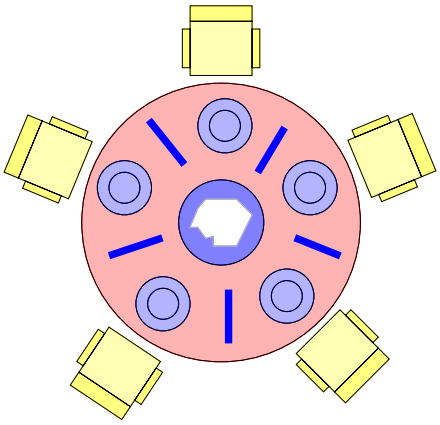
- Einordnung:



  - Verhalten von Aktivitätsträgern / Prozessen

1 Motivation

- Beispiel: die fünf Philosophen am runden Tisch

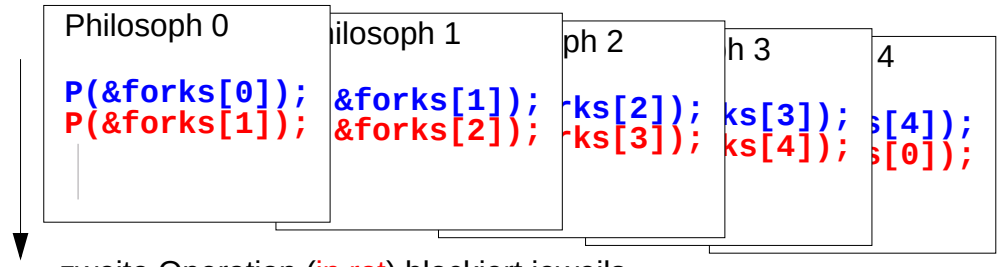


  - Philosophen denken oder essen
    "The life of a philosopher consists of an alternation of thinking and eating." (Dijkstra, 1971)
  - zum Essen benötigen sie zwei Gabeln, die jeweils zwischen zwei benachbarten Philosophen abgelegt sind

- Philosophen können verhungern, wenn sie sich „dumm“ anstellen.

1 Motivation (2)

- Problem der Verklemmung *(Deadlock)*
  - alle Philosophen nehmen gleichzeitig die linke Gabel auf und versuchen dann die rechte Gabel aufzunehmen



zweite Operation (in rot) blockiert jeweils

  - System ist **verklemmt:** Philosophen warten alle auf ihre Nachbarn
- Problemkreise:
  - Vermeidung und Verhinderung von Verklemmungen
  - Erkennung und Erholung von Verklemmungen

2 Betriebsmittelbelegung

- Betriebsmittel
  - CPU, Drucker, Geräte (Platten, CD-ROM, Floppy, Audio, usw.)
  - nur elektronisch vorhandene Betriebsmittel der Anwendung oder des Betriebssystems, z.B. Gabeln der Philosophen
- Unterscheidung von Typ und Instanz
  - Typ definiert ein Betriebsmittel eindeutig
  - Instanz ist eine Ausprägung des Typs
    (die Anwendung benötigt eine Instanz eines best. Typs, egal welche)
    - **CPU:** Anwendung benötigt eine von mehreren gleichartigen CPUs
    - **Drucker:** Anwendung benötigt einen von mehreren gleichen Druckern (falls Drucker nicht austauschbar und gleichwertig, so handelt es sich um verschiedene Typen)
    - **Gabeln:** jede Gabel ist ein eigener Betriebsmitteltyp

2.1 Belegung

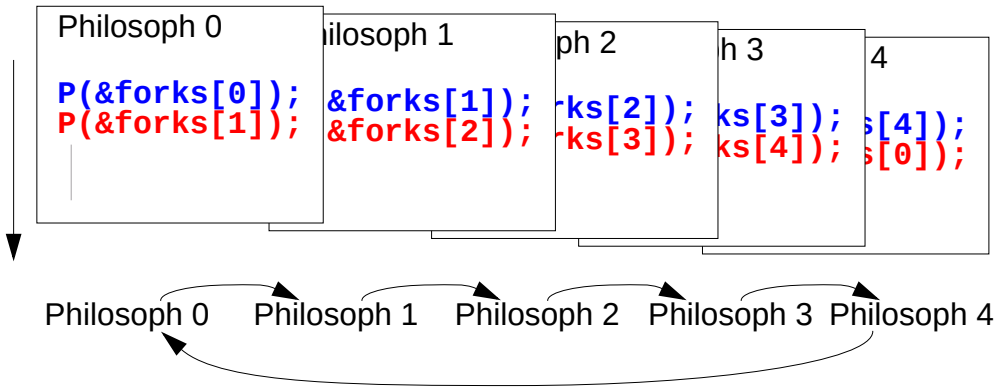
- Belegung erfolgt in drei Schritten
  - ◆ Anfordern des Betriebsmittels
    - blockiert evtl. falls Betriebsmittel nur exklusiv benutzt werden kann
    - **Gabel:** nur exklusiv
    - **Bildschirmausgabe:** exklusiv oder nicht-exklusiv
  - ◆ Nutzen des Betriebsmittels
    - **Gabel:** Philosoph kann essen
    - **Drucker:** Anwendung kann drucken
  - ◆ Freigeben des Betriebsmittels
    - **Gabel:** Philosoph legt Gabel wieder zwischen die Teller

2.2 Voraussetzungen für Verklemmungen

- Vier notwendige Bedingungen
  - ◆ *Exklusive Belegung*
    Mindestens ein Betriebsmitteltyp muss nur exklusiv belegbar sein.
  - ◆ *Nachforderungen von Betriebsmittel möglich*
    Es muss einen Prozess geben, der bereits Betriebsmittel hält, und ein neues Betriebsmittel anfordert.
  - ◆ *Kein Entzug von Betriebsmitteln möglich*
    Betriebsmittel können nicht zurückgefordert werden bis der Prozess sie wieder freigibt.
  - ◆ *Zirkuläres Warten*
    Es gibt einen Ring von Prozessen, in dem jeder auf ein Betriebsmittel wartet, das der Nachfolger im Ring besitzt.

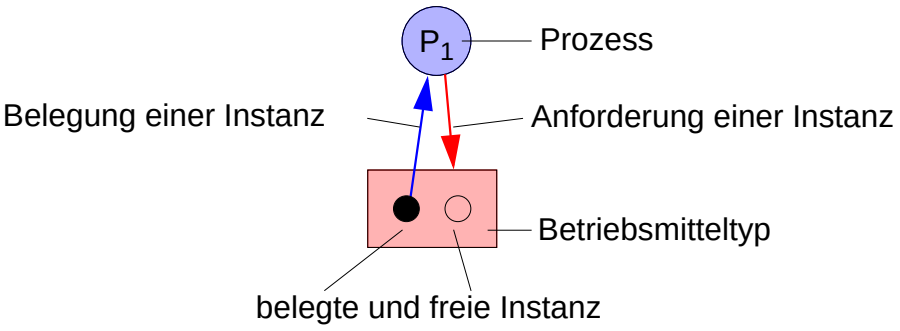
2.2 Voraussetzungen für Verklemmung (2)

- Beispiel: fünf Philosophen
  - Exklusive Belegung: **ja**
  - Nachforderungen von Betriebsmittel möglich: **ja**
  - Entzug von Betriebsmitteln: **nicht vorgesehen**
  - Zirkuläres Warten: **ja**



2.3 Betriebsmittelgraphen

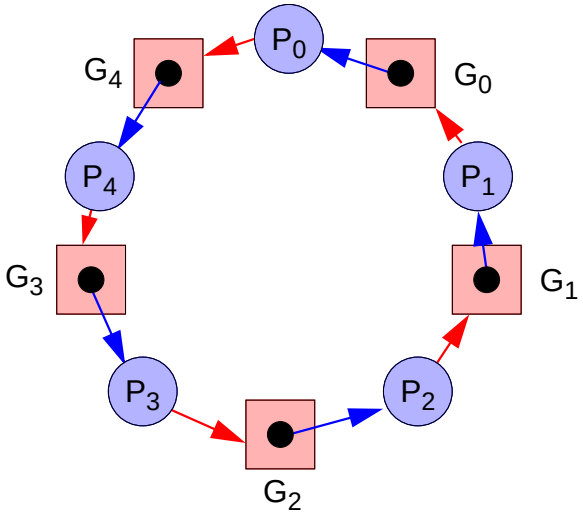
- Veranschaulichung der Belegung und Anforderung durch Graphen (nur exklusive Belegungen)



- Regeln:
  - kein Zyklus im Graph → keine Verklemmung
  - Zyklus im Graph → Verklemmung
  - nur jeweils eine Instanz pro Betriebsmitteltyp und Zyklus → Verklemmung

2.3 Betriebsmittelgraphen (2)

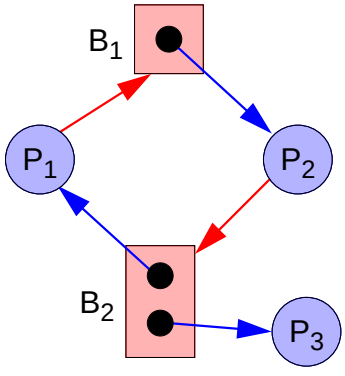
- Beispiel: fünf Philosophen



◆ Zyklus und jeder Betriebsmitteltyp hat nur eine Instanz ➔ Verklemmung

2.3 Betriebsmittelgraphen (3)

- Beispiel mit Zyklus und ohne Verklemmung



◆ Prozess 3 kann seine Instanz vom Betriebsmitteltyp B_2 wieder zurückgeben und den Zyklus damit auflösen

3 Vermeidung von Verklemmungen

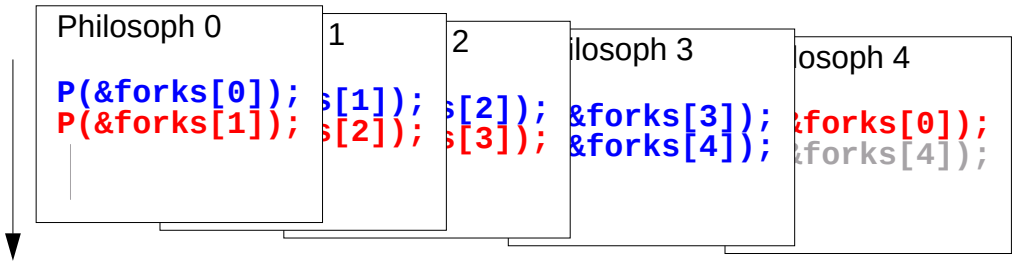
- Ansatz: Vermeidung der notwendigen Bedingungen für Verklemmungen
  - *Exklusive Belegung:*
    oft nicht vermeidbar
  - *Nachforderungen von Betriebsmittel möglich:*
    alle Betriebsmittel müssen auf einmal angefordert werden
    - ungenutzte aber belegte Betriebsmittel vorhanden
    - Aushungerung möglich: ein anderer Prozess hält immer das nötige Betriebsmittel belegt

3 Vermeidung von Verklemmungen (2)

- *Kein Entzug von Betriebsmitteln möglich:*
  Entzug von Betriebsmitteln erlauben
  - bei neuer Belegung werden alle gehaltenen Betriebsmittel freigegeben und mit der neuen Anforderung zusammen wieder angefordert
  - während ein Prozess wartet, werden seine bereits belegten Betriebsmittel anderen Prozessen zur Verfügung gestellt
  - möglich für CPU oder Speicher jedoch nicht für Drucker, Bandlaufwerke oder ähnliche
- *Zirkuläres Warten:* Vermeidung von Zyklen
  - Totale Ordnung auf Betriebmitteltypen

3 Vermeidung von Verklemmungen (3)

- Anforderungen nur in der Ordnungsreihenfolge erlaubt

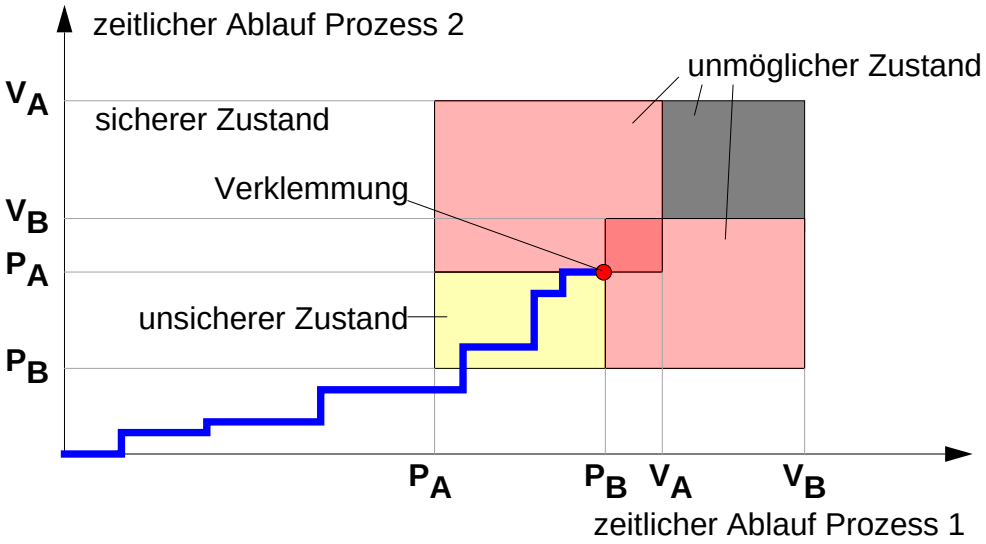


z.B. Gabeln: geordnet nach Gabelnummer

- Bei neuer Anforderung wird geprüft, ob letzte Anforderung kleiner bzgl. der totalen Ordnung war (Instanzen gleichen Typs müssen gleichzeitig angefordert werden); sonst: Abbruch mit Fehlermeldung
- Philosoph 4 bekäme eine Fehlermeldung, wenn er in der obigen Situation zuerst Gabel 4 und dann Gabel 0 anfordert: Rückgabe und neuer Versuch

4 Verhinderung von Verklemmungen

- Annahme: es ist bekannt, welche Betriebsmittel ein Prozess brauchen wird (hier je zwei binäre Semaphore A und B)
  - ◆ Betriebssystem überprüft System auf unsichere Zustände



4.1 Sichere und unsichere Zustände

- Sicherer Zustand
  - Es gibt eine Sequenz, in der die vorhandenen Prozesse abgearbeitet werden können, so dass ihre Anforderungen immer befriedigt werden können.
  - Sicherer Zustand erlaubt immer eine verklemmungsfreie Abarbeitung
- Unsicherer Zustand
  - Es gibt keine solche Sequenz.
  - Verklemmungszustand ist ein unsicherer Zustand
  - Ein unsicherer Zustände führt zwangsläufig zur Verklemmung, wenn die Prozesse ihre angenommenen Betriebsmittel wirklich anfordern bevor sie von anderen Prozessen wieder freigegeben werden.

4.1 Sichere und unsichere Zustände (2)

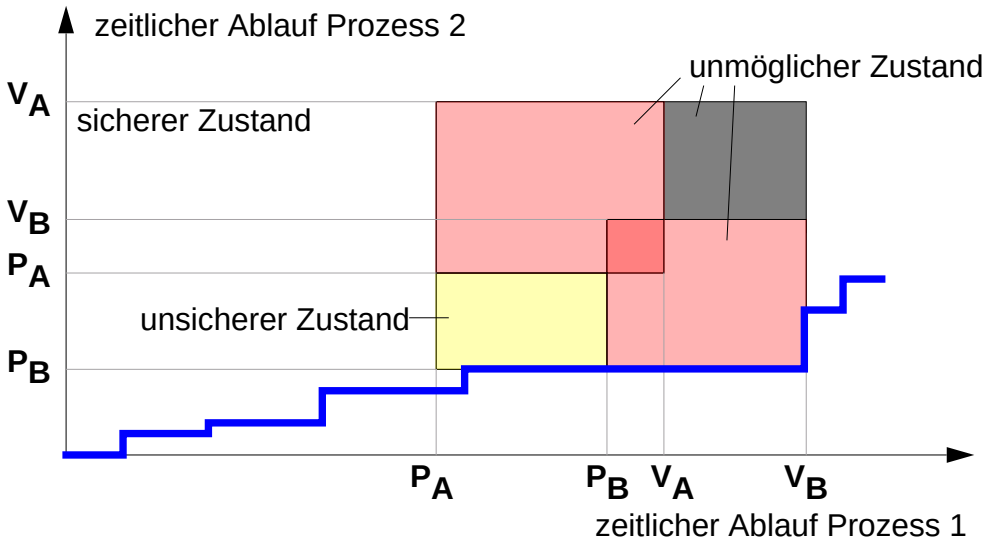
- Beispiel:
  - 12 Magnetbandlaufwerke vorhanden
  - P_0 braucht (bis zu) 10 Laufwerke
  - P_1 braucht (bis zu) 4 Laufwerke
  - P_2 braucht (bis zu) 9 Laufwerke
  - Aktuelle Situation: P_0 hat 5, P_1 hat 2 und P_2 hat 2 Laufwerke
  - Zustand sicher?
  - Aktuelle Situation: P_0 hat 5, P_1 hat 2 und P_2 hat 3 Laufwerke
  - Zustand sicher?

4.1 Sichere und unsichere Zustände (3)

- Verhinderung von Verklemmungen
  - Verhinderung von unsicheren Zuständen
  - Anforderungen blockieren, falls sie in einen unsicheren Zustand führen würden
- Beispiel von Folie H.page 17:
  - Zustand: P_0 hat 5, P_1 hat 2 und P_2 hat 2 Laufwerke
  - P_2 fordert ein zusätzliches Laufwerk an
  - Belegung würde in unsicheren Zustand führen: P_2 muss warten
- Verhinderung von unsicheren Zuständen schränkt Nutzung von Betriebsmitteln ein
  - verhindert aber Verklemmungen

4.1 Sichere und unsichere Zustände (4)

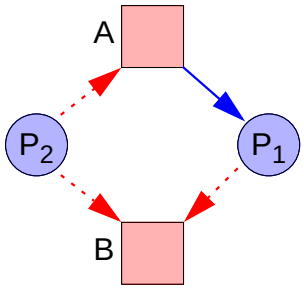
- Beispiel von Folie H.page 15:



  - Prozess 2 darf **P_B** nicht durchführen und muss warten

4.2 Betriebsmittelgraph

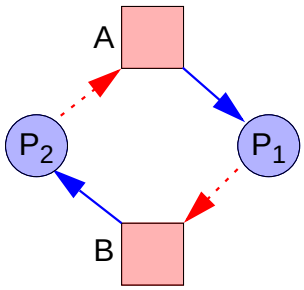
- Annahme: eine Instanz pro Betriebsmitteltyp
  - Einsatz von Betriebsmittelgraphen zur Erkennung unsicherer Zustände



  - zusätzliche Kanten zur Darstellung möglicher Anforderungen (Ansprüche, *Claims*)
  - Anspruchskanten werden gestrichelt dargestellt und bei Anforderung in Anforderungskanten umgewandelt
  - Anforderung und Belegung von B durch P_2 führt in einen unsicheren Zustand (siehe Beispiel von Folie H.15)

4.2 Betriebsmittelgraph (2)

- Erkennung des unsicheren Zustands an Zyklen im erweiterten Betriebsmittelgraph
  - Anforderung und Belegung von B durch P_2 führt zu:



  - Zyklenerkennung hat einen Aufwand von $O(n^2)$
- Betriebsmittelgraph nicht anwendbar bei mehreren Instanzen eines Betriebsmitteltyps

4.3 Banker's Algorithm

- Erkennung unsicherer Zustände bei mehreren Instanzen pro Betriebsmitteltyp
- Annahmen:
  - m Betriebsmitteltypen; Typ i verfügt über b_i Instanzen
  - n Prozesse
- Definitionen
  - B ist der Vektor $(b_1, b_2, \ldots b_m)$ der vorhandenen Instanzen
  - R ist der Vektor $(r_1, r_2, \ldots r_m)$ der noch verfügbaren Restinstanzen
  - C_j sind die Vektoren $(c_{j,1}, c_{j,2}, \ldots c_{j,m})$ der aktuellen Belegung durch den Prozess j
- Es gilt: $\sum_{j=1}^{n} c_{j,i} + r_i = b_i$ für alle $1 \le i \le m$

4.3 Banker's Algorithm (2)

- Weitere Definitionen
  - M_j sind die Vektoren $(m_{j,1}, m_{j,2}, \ldots m_{j,m})$ der bekannten maximalen Belegung der Betriebsmittel 1 bis m durch den Prozess j
  - zwei Vektoren A und B stehen in der Relation A ≤ B, falls die Elemente der Vektoren jeweils paarweise in der gleichen Relation stehen
    z.B. $(1, 2, 3) \le (2, 2, 4)$

4.3 Banker's Algorithm (3)

- Algorithmus
  1. alle Prozesse sind zunächst unmarkiert
  2. wähle einen nicht markierten Prozess *j*, so dass $M_j - C_j \leq R$
     (Prozess ist ohne Verklemmung ausführbar, selbst wenn er alles anfordert, was er je brauchen wird)
  3. falls ein solcher Prozess *j* existiert, addiere C_j zu *R*, markiere Prozess *j* und beginne wieder bei Punkt (2)
     (Bei Terminierung wird der Prozess alle Betriebsmittel freigeben)
  4. falls ein solcher Prozess nicht existiert, terminiere Algorithmus

◆ Sind alle Prozesse markiert, ist das System in einem sicheren Zustand.

4.4 Beispiel

- Beispiel:
  - ◆ 12 Magnetbandlaufwerke vorhanden
  - ◆ P_0 braucht (bis zu) 10 Laufwerke
  - ◆ P_1 braucht (bis zu) 4 Laufwerke
  - ◆ P_2 braucht (bis zu) 9 Laufwerke
  - ◆ Aktuelle Situation: P_0 hat 5, P_1 hat 2 und P_2 hat 3 Laufwerke
- Belegung der Datenstrukturen
  - ◆ *m = 1*
  - ◆ *n = 3*
  - ◆ *B = (12)*
  - ◆ *R = (2)*
  - ◆ $C_0 = (5), C_1 = (2), C_2 = (3)$
  - ◆ $M_0 = (10), M_1 = (4), M_2 = (9)$

4.4 Beispiel (2)

- Anwendung des Banker's Algorithm
  - wähle einen nicht markierten Prozess j, so dass $M_j - C_j \leq R$
    → P_1
  - $R := R + C_1$ → $R = (4)$
  - wähle einen nicht markierten Prozess j, so dass $M_j - C_j \leq R$
    → kein geeigneter Prozess vorhanden
  - Zustand ist unsicher

5 Erkennung von Verklemmungen

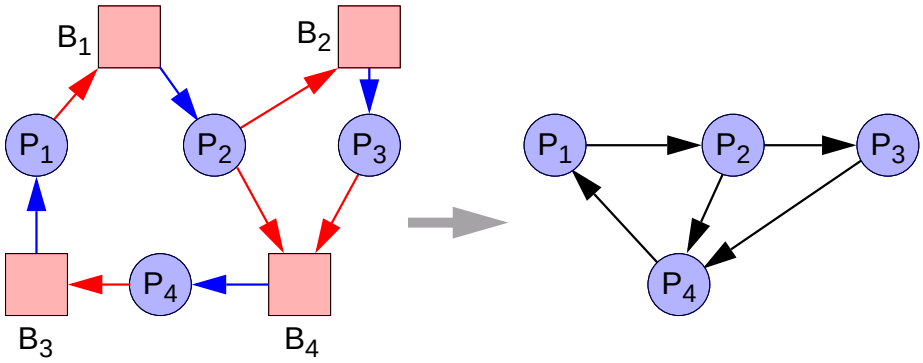
- Systeme ohne Mechanismen zur Vermeidung oder Verhinderung von Verklemmungen
  - Verklemmungen können auftreten
  - Verklemmung sollte als solche erkannt werden
  - Auflösung der Verklemmung sollte eingeleitet werden (Algorithmus nötig)

5.1 Wartegraphen

- Annahme: nur eine Instanz pro Betriebsmitteltyp
  - Einsatz von Wartegraphen, die aus dem Betriebsmittelgraphen gewonnen werden können

5.1 Wartegraphen (2)

- Wartegraphen
  - Betriebsmittel und Kanten werden aus Betriebsmittelgraph entfernt
  - zwischen zwei Prozessen wird eine „wartet auf“-Kante eingeführt, wenn es Kanten vom ersten Prozess zu einem Betriebsmittel und von diesem zum zweiten Prozess gibt



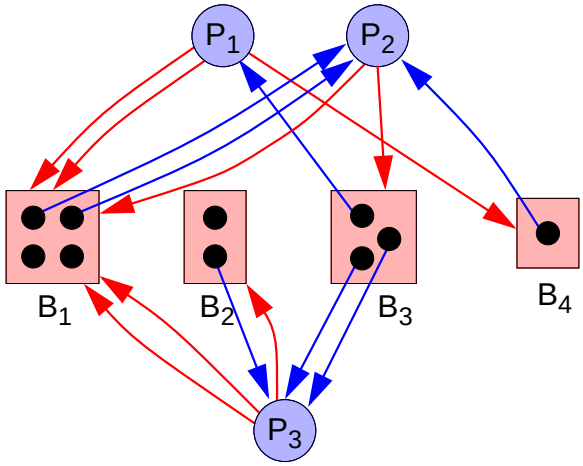
Betriebsmittelgraph Wartegraph

5.1 Wartegraphen (3)

- Erkennung von Verklemmungen
  - Wartegraph enthält Zyklen: System ist verklemmt
- Betriebsmittelgraph nicht für Systeme geeignet, die mehrere Instanzen pro Betriebsmitteltyp zulassen

5.2 Erkennung durch graphische Reduktion

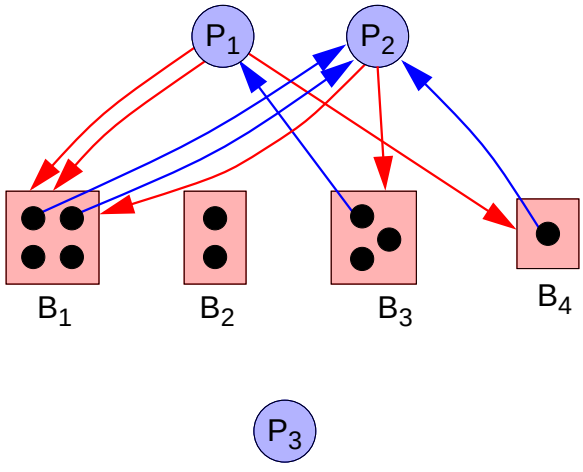
- Betriebsmittelgraph des Beispiels



- ◆ Auswahl eines Prozesses für den Anforderungen erfüllbar: nur P_3 möglich
- ◆ Löschen aller Kanten des Prozesses

5.2 Erkennung durch graphische Reduktion (2)

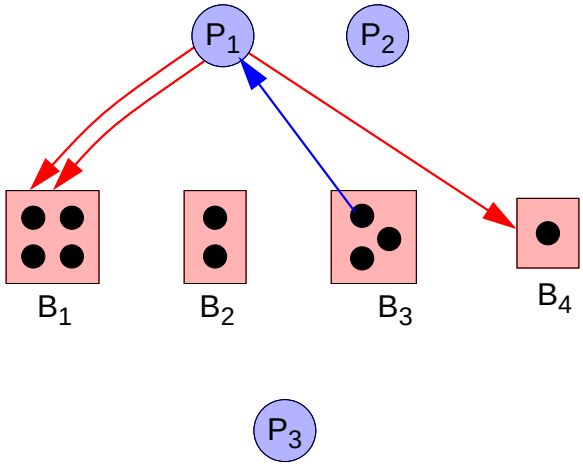
- Betriebsmittelgraph des Beispiels (1. Reduktion)



- ◆ Auswahl eines Prozesses für den Anforderungen erfüllbar: nur P_2 möglich
- ◆ Löschen aller Kanten des Prozesses

5.2 Erkennung durch graphische Reduktion (3)

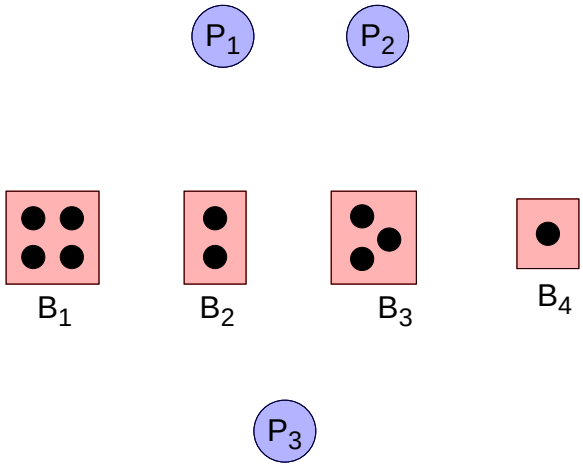
- Betriebsmittelgraph des Beispiels (2. Reduktion)



- Auswahl eines Prozesses für den Anforderungen erfüllbar: P_1
- Löschen aller Kanten des Prozesses

5.2 Erkennung durch graphische Reduktion (4)

- Betriebsmittelgraph des Beispiels (3. Reduktion)



- es bleiben keine Prozesse mit Anforderungen übrig ➔ keine Verklemmung
- übrig bleibende Prozesse sind verklemmt und in einem Zyklus

5.3 Erkennung durch Reduktionsverfahren

- Annahmen:
  - m Betriebsmitteltypen; Typ i verfügt über b_i Instanzen
  - n Prozesse
- Definitionen
  - B ist der Vektor $(b_1, b_2, ... b_m)$ der vorhandenen Instanzen
  - R ist der Vektor $(r_1, r_2, ... r_m)$ der noch verfügbaren Restinstanzen
  - C_j sind die Vektoren $(c_{j,1}, c_{j,2}, ... c_{j,m})$ der aktuellen Belegung durch den Prozess j
- Es gilt: $\sum_{j=1}^{n} c_{j,i} + r_i = b_i$ für alle $1 \le i \le m$

5.3 Erkennung durch Reduktionsverfahren (2)

- Weitere Definitionen
  - A_j sind die Vektoren $(a_{j,1}, a_{j,2}, ... a_{j,m})$ der aktuellen Anforderungen durch den Prozess j
  - zwei Vektoren A und B stehen in der Relation A ≤ B, falls die Elemente der Vektoren jeweils paarweise in der gleichen Relation stehen
- Algorithmus
  1. alle Prozesse sind zunächst unmarkiert
  2. wähle einen Prozess j, so dass $A_j \le R$
     (Prozess ist ohne Verklemmung ausführbar)
  3. falls ein solcher Prozess j existiert, addiere C_j zu R, markiere Prozess j und beginne wieder bei Punkt (2)
     (Bei Terminierung wird der Prozess alle Betriebsmittel freigeben)
  4. falls ein solcher Prozess nicht existiert, terminiere Algorithmus
  - alle nicht markierten Prozesse sind an einer Verklemmung beteiligt

5.3 Erkennung durch Reduktionsverfahren (3)

- Beispiel
  - $m = 4$; $B = (4, 2, 3, 1)$
  - $n = 3$; $C_1 = (0, 0, 1, 0)$; $C_2 = (2, 0, 0, 1)$; $C_3 = (0, 1, 2, 0)$
  - daraus ergibt sich $R = (2, 1, 0, 0)$
  - Anforderungen der Prozesse lauten:
    $A_1 = (2, 0, 0, 1)$; $A_2 = (1, 0, 1, 0)$; $A_3 = (2, 1, 0, 0)$
- Ablauf
  - Auswahl eines Prozesses: Prozess 3, da $A_3 \leq R$; markiere Prozess 3
  - Addiere C_3 zu R: neues $R = (2, 2, 2, 0)$
  - Auswahl eines Prozesses: Prozess 2, da $A_2 \leq R$; markiere Prozess 2
  - Addiere C_2 zu R: neues $R = (4, 2, 2, 1)$
  - Auswahl eines Prozesses: Prozess 1, da $A_1 \leq R$; markiere Prozess 1
  - kein Prozess mehr unmarkiert: keine Verklemmung

5.4 Einsatz der Verklemmungserkennung

- Wann sollte Erkennung ablaufen?
  - Erkennung ist aufwendig (Aufwand $O(n^2)$ bei Zyklenerkennung)
  - Häufigkeit von Verklemmungen eher gering
  - zu häufig: Verschwendung von Ressourcen zur Erkennung
  - zu selten: Betriebsmittel werden nicht optimal genutzt, Anzahl der verklemmten Prozesse steigt
- Möglichkeiten:
  - Erkennung, falls eine Anforderung nicht sofort erfüllt werden kann
  - periodische Erkennung (z.B. einmal die Stunde)
  - CPU Auslastung beobachten; falls Auslastung sinkt, Erkennung starten

5.5 Erholung von Verklemmungen

- Verklemmung erkannt: Was tun?
  - Operateur benachrichtigen; manuelle Beseitigung
  - System erholt sich selbst
- Abbrechen von Prozessen (terminierte Prozesse geben ihre Betriebsmittel wieder frei)
  - alle verklemmten Prozesse abbrechen (großer Schaden)
  - einen Prozess nach dem anderen abbrechen bis Verklemmung behoben (kleiner Schaden aber rechenzeitintensiv)
  - mögliche Schäden:
    - Verlust von berechneter Information
    - Dateninkonsistenzen

5.5 Erholung von Verklemmungen (2)

- Entzug von Betriebsmitteln
  - Aussuchen eines „Opfer“-Prozesses (Aussuchen nach geringstem entstehendem Schaden)
  - Entzug der Betriebsmittel und Zurückfahren des „Opfer“-Prozesses (Prozess wird in einen Zustand zurückgefahren, der unkritisch ist; benötigt Checkpoint oder Transaktionsverarbeitung)
  - Verhinderung von Aushungerung (es muss verhindert werden, dass immer derselbe Prozess Opfer wird und damit keinen Fortschritt mehr macht)

6 Kombination der Verfahren

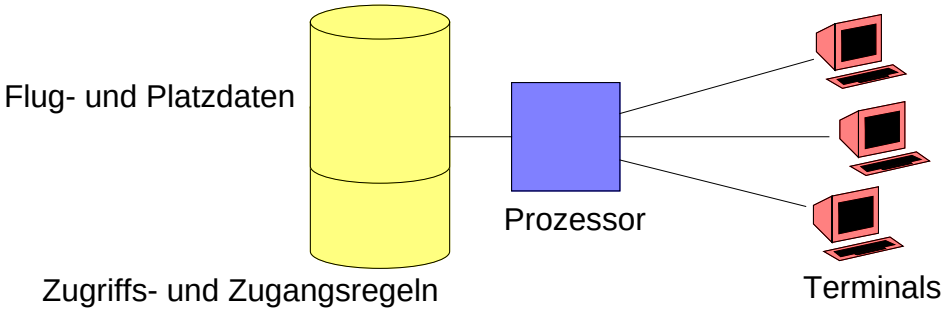
- Einsatz verschiedener Verfahren für verschiedene Betriebsmittel
  - Interne Betriebsmittel:
    Verhindern von Verklemmungen durch totale Ordnung der Betriebsmittel (z.B. IBM Mainframe-Systeme)
  - Hauptspeicher:
    Verhindern von Verklemmungen durch Entzug des Speichers (z.B. durch Swap-Out)
  - Betriebsmittel eines Jobs:
    Angabe der benötigten Betriebsmittel beim Starten; Einsatz der Vermeidungsstrategie durch Feststellen unsicherer Zustände
  - Hintergrundspeicher (Swap-Space):
    Vorausbelegung des Hintergrundspeichers

I Datensicherheit und Zugriffschutz

I Datensicherheit und Zugriffsschutz

1 Problemstellung

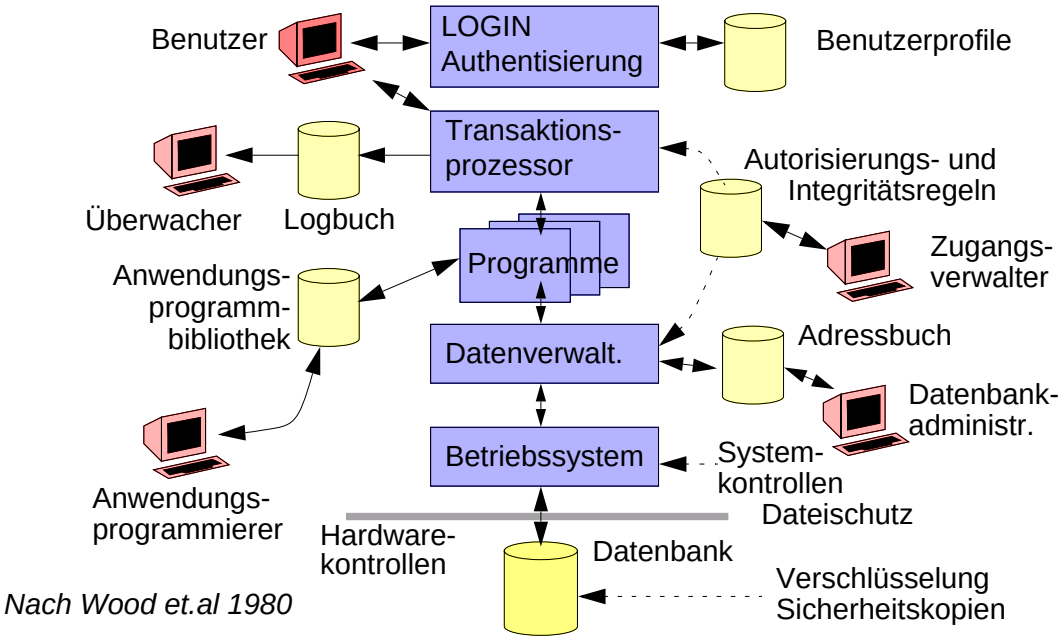
- Beispiel: Zugang zu einer Datenbank zur Flugreservierung und -buchung



◆ Was sind mögliche Beeinträchtigungen der Datensicherheit?

1 Problemstellung (2)

- Überprüfungen beim Transaktionsbetrieb (Datenbankanwendung)



Nach Wood et.al 1980

1 Problemstellung (3)

- Illegaler Datenzugriff
  - Daten sind zugreifbar, die vertraulich behandelt werden sollen
- Illegales Löschen von Daten
  - Kein Zugriff, aber Daten werden gelöscht
- Illegales Manipulieren von Daten
  - Daten werden in böswilliger Absicht verändert
- Zerstörung von Rechensystemen
  - physisches Zerstören von Teilen der Rechenanlage

1.1 Umgebung der Rechenanlage

- Naturkatastrophen
  - Erdbeben, Vulkanausbrüche etc. können Rechenanlage und Datenbestand zerstören
- Unfälle
  - Gasexplosion, Kühlwasserlecks in der Klimaanlage oder Ähnliches zerstören Rechner und Daten
- Böswillige Angriffe
  - Zerstörung der Rechenanlage und des Datenbestands durch Sabotage (Bombenanschlag, Brandanschlag etc.)
- Unbefugter Zutritt zu den Räumen des Rechenzentrums
  - Diebstahl von Datenträgern
  - Zerstörung von Daten
  - Zugang zu vertraulichen Daten

1.2 Systemsoftware

- Versagen der Schutzmechanismen
  - System lässt Unbefugte auf Daten zugreifen oder Operationen ausführen
- Durchsickern von Informationen
  - Anwender können anhand scheinbar unauffälligen Systemverhaltens Rückschlüsse auf vertrauliche Daten ziehen *(Covert channels)*
- Beispiel: verschlüsselt abgespeicherte Passwörter sind zugänglich
  - Entschlüsselungsversuch der Passwörter außerhalb der Rechenanlage
  - "Wörterbuchattacke": Raten von Passwörtern möglich

1.3 Systemprogrammierer

- Umgehen oder Abschalten der Schutzmechanismen
- Installation eines unsicheren Systems
  - erlaubt dem Systemprogrammierer die Schutzmechanismen von außen zu umgehen
- Fehler beim Nutzen von Bibliotheksfunktionen innerhalb sicherheitskritischer Programme
  - S-Bit Programme unter UNIX laufen mit der Benutzerkennung des Dateibesitzers, nicht unter der des Aufrufers
    - Fehlerhafte S-Bit Programme können zur Ausführung von Code unter einer fremden Benutzerkennung gebracht werden
    - S-Bit Programme mit "root-Rechten" besonders gefährlich

1.3 Systemprogrammierer (2)

- Buffer-Overflow Fehler bei Funktionen
  `gets(), strcpy(), strcat(), sprintf( "%s",..)`
  - Zu lange Eingaben überschreiben den Stackspeicher des Prozessors
  - Mit genauer Kenntnis ist das Ausführen beliebigen Codes erzwingbar

- Fehlerhafte Parameterprüfung beim Aufruf von Funktionen
  `system()`
- Beispiel: Löschen einer Datei
  - Name der Datei wurde in Variable **`file`** eingelesen
  - Aufruf von **`system`** mit Parameter **`strcat( "rm ", file)`**
  - Gibt man für den Dateinamen den String
    `"fn ; xterm -display myhost:0 &"`
    ein, bekommt man ein Fenster auf der aufgebrochenen Maschine

1.4 Rechnerhardware

- Versagen der Schutzmechanismen
  - erlauben nicht-autorisierten Zugriff
- Fehlerhaft Befehlsausführung
  - Zerstörung von wichtigen Daten
- Abstrahlungen
  - erlaubt Ausspähen von Daten

1.5 Datenbasis

- Falsche Zugriffsregeln
  - erlauben nicht-autorisierten Zugriff

1.6 Operateur

- Kopieren vertraulicher Datenträger
- Diebstahl von Datenträgern
- Initialisierung mit unsicherem Zustand
  - Operateur schaltet beispielsweise Zugriffskontrolle ab
- Nachlässige Rechnerwartung
  - Nachbesserungen der Systemsoftware *(Patches)* werden nicht eingespielt
    - Sicherheitslücken werden nicht gestopft

1.7 Sicherheitsbeauftragter

- Fehlerhafte Spezifikation der Sicherheitspolitik
  - dadurch Zugang für Unbefugte zu vertraulichen Daten oder
  - Änderungen von Daten durch Unbefugte möglich
- Unterlassene Auswertung von Protokolldateien
  - Einbrüche und mögliche Sicherheitslücken werden nicht rechtzeitig entdeckt

1.8 Kommunikationssystem

- Abhören der Kommunikationsleitungen *(Sniffing)*
  - z.B. Telefonverbindung bei Modemnutzung oder serielle Schnittstellen
  - z.B. Netzwerkverkehr auf einem Netzwerkstrang
  - Ermitteln von Passwörtern und Benutzerkennungen
    - manche Dienste übertragen Passwörter im Klartext (z.B. ftp, telnet, rlogin)
  - Zugriff auf vertrauliche Daten
  - unbefugte Datenveränderungen
    - Verfälschen von Daten
    - Übernehmen von bestehenden Verbindungen *(Hijacking)*
    - Vorspiegeln falscher Netzwerkadressen *(Spoofing)*

1.8 Kommunikationssystem (2)

- Illegale Nutzung von Diensten über das Netzwerk
  - Standardsysteme bieten eine Menge von Diensten an (z.B. ftp, telnet, rwho u.a.)
  - Sicherheitslücken von Diensten werden publik gemacht und sind auch von "dummen" Hackern nutzbar *(Exploit scripts)*
    http://www.rootshell.org
  - Auch bei temporär am Netzwerk angeschlossenen Computern eine Gefahr
    - z.B. Linux-Maschine mit PPP-Verbindung an das Uni-Netz
    - Voreinstellungen der Standardinstallation meist unsicher

1.9 Terminal

- Ungeschützter Zugang zum Terminal
  - Nutzen einer fremden Benutzerkennung
  - Zugriff auf vertrauliche Daten
  - unbefugte Datenveränderungen

1.10 Benutzer

- Nutzen anderer Kennungen
  - erlauben nicht-autorisierten Zugriff
  - unbefugte Datenveränderungen
  - unbefugte Weitergabe von Informationen
- Einbruch von Innen
  - leichterer Zugang zu möglichen Sicherheitslöchern (z.B. bei Diensten)

1.11 Anwendungsprogrammierung

- Nichteinhalten der Spezifikation
  - Umgehen der Zugriffskontrollen
- Einfügen von „bösartigen“ Befehlsfolgen
  - *Back door:* Hintertür gibt dem Programmierer im Betrieb Zugang zu vertraulichen Daten oder illegalen Operationen
  - *Trojan horse:* Unter bestimmten Bedingungen werden illegale Operationen ohne Trigger von außen angestoßen

1.12 "Tracker Queries"

- Beispiel: Datenbanksysteme
  - Zugriff auf Einzelinformationen ist verboten (Vertraulichkeit)
  - statistische Informationen sind erlaubt
- Grenzen möglicher Sicherheitsmaßnahmen:
  Zugriff auf Einzelinformationen dennoch möglich
  - geeignete Anfragen kombinieren *(Tracker queries)*
- Beispiel: Gehaltsdatenbank

1.12 "Tracker Queries" (2)

- Tabelle der Datenbankeinträge:

Nr.	Name	Geschl.	Fach	Stellung	Gehalt	Spenden
1	Albrecht	m	Inf.	Prof.	60.000	150
2	Bergner	m	Math.	Prof.	45.000	300
3	Cäsar	w	Math.	Prof.	75.000	600
4	David	w	Inf.	Prof.	45.000	150
4	Engel	m	Stat.	Prof.	54.000	0
5	Frech	w	Stat.	Prof.	66.000	450
6	Groß	m	Inf.	Angest.	30.000	60
8	Hausner	m	Math.	Prof.	54.000	1500
9	Ibel	w	Inf.	Stud.	9.000	30
10	Jost	m	Stat.	Angest.	60.000	45
11	Knapp	w	Math.	Prof.	75.000	300
12	Ludwig	m	Inf.	Stud.	9.000	0

1.12 "Tracker Queries" (3)

- Anfragen und Antworten:
  - Anzahl('w'): 5
  - Anzahl('w' und (nicht 'Inf' oder nicht 'Prof.')): 4
  - mittlere Spende('w'): 306
  - mittlere Spende('w' und (nicht 'Inf.' oder nicht 'Prof.')): 345
- Berechnung:
  - Spende('David'): 306 * 5 - 345 * 4 =
1530 - 1380 =
150

2 Zugriffslisten

- Identifikation von Subjekten, Objekten und Berechtigungen
  - Subjekt: Person oder Benutzerkennung im System
(repräsentiert jemanden, der Aktionen ausführen kann)
  - Objekt: Komponente des Systems
(repräsentiert Ziel einer Aktion)
  - Berechtigung: z.B. Leseberechtigung auf einer Datei
(repräsentiert die Erlaubnis für die Ausführung einer Aktion)
- Erfassung der Berechtigungen in einer Subjekt-Objekt-Matrix:
Zugriffsliste *(Access control list, ACL)*

2.1 Beispiel für Zugriffslisten

- Personaldatensatz
  - besteht aus: Name, Abteilung, Personalnummer, Lohn- oder Gehaltsgruppe
- Personaldateien (Objekte)
  - D_{LA}: Personaldaten der leitenden Angestellten
  - D_{AN}: Personaldaten der sonstigen Angestellten
  - D_{AR}: Personaldaten der Arbeiter
- Prozeduren (gehören zu den Aktionen)
  - R_{LA}: Lesen von Pers.-Nr. und Lohn-/Gehaltsgr. aus D_{LA}
  - $R_{AN/AR}$: Lesen von Pers.-Nr. und Lohn-/Gehaltsgr. aus D_{AN} oder D_{AR}
  - R_{post}: Lesen von Name, Abteilung und Pers.-Nr.

2.1 Beispiel für Zugriffslisten (2)

- Benutzer (Subjekte)
  - S_{pers}: Leiter des Personalbüros
    - Besitzer aller Dateien und Prozeduren
    - Lese- und Schreibrecht für alleDateien
    - Aufrufrecht für alle Prozeduren
  - S_{stellv}: Sachbearb. leitende Angestellte, stellvertr. Leiter Personalbüro
    - Lese- und Schreibrecht für D_{AN} und D_{AR}
    - Aufrufrecht für R_{LA}
  - S_{sach}: Sachbearbeiter Angestellte u. Arbeiter
    - Aufrufrecht für $R_{AN/AR}$
  - S_{post}: Poststelle
    - Aufrufrecht für R_{post} auf alle Dateien

2.1 Beispiel für Zugriffslisten (3)

- Berechtigungen werden in Matrix ausgedrückt:

	D_{LA}	D_{AN}	D_{AR}	R_{LA}	$R_{AN/AR}$	R_{post}
S_{pers}	O, R, W	O, R, W	O, R, W	O, I	O, I	O, I
S_{stellv}		R, W	R, W	I		
S_{sach}					I	
S_{post}						I
R_{LA}	R					
$R_{AN/AR}$		R	R			
R_{post}	R	R	R			

- O = *Owner;* Besitzer der Datei oder Prozedur
- R = *Read;* volle Leseberechtigung
- W = *Write;* volle Schreibberechtigung
- I = *Invoke;* Aufrufberechtigung

2.2 Beispiel: UNIX

- Zugriffslisten für
  - Dateien und Geräte
  - Shared-Memory-Segmente
  - Message-Queues
  - Semaphore
  - etc.
- Berechtigungen:
  - Lesen *(read)*, Schreiben *(write)*, Ausführen *(execute)*
  - für Besitzer, Gruppe und alle anderen unterscheidbar
- Subjekte:
  - Prozesse
  - Besitzer (Benutzer) und Zugehörigkeit zu einer oder mehreren Gruppen

2.2 Beispiel: UNIX

- Superuser
  - Benutzer *root* hat automatisch alle Zugriffsrechte
- S-Bit–Programme
  - S-Bit ist ein besonderes Recht auf der Binärdatei des Programms
  - Besitzer der Datei wird bei der Ausführung auch Besitzer des Prozess (sonst wird Aufrufer Besitzer des Prozess)
- ★ Vorteil
  - Bereitstellen von Prozessen, die kontrolliert Aufrufern höhere Zugriffsberechtigungen erlauben
- ▲ Nachteil
  - Fehler im Prozess gibt Aufrufer volle Rechte des Programmbesitzers
  - fatal, falls das Programm *root* gehört

2.3 Implementierung

- Globale Tabelle/Matrix
  - System hält eine Datenstruktur und prüft im betreffenden Eintrag die Berechtigungen
  - Tabelle üblicherweise recht groß: passt evtl. nicht in den Speicher
- Zugriffslisten an den Objekten
  - jedes Objekt hält eine Liste der Berechtigungen (z.B. Unix Datei: Inode)
  - verringert üblicherweise den Platzbedarf für die Einträge (unnötige Felder der Matrix werden nicht repräsentiert)
- Zugriffslisten an den Subjekte
  - jedes Subjekt hält eine Liste von Objekten und den Berechtigungen, die das Subjekt für das Objekt hat
  - ähnlich Capabilities

3 Schutz durch Speicherverwaltung

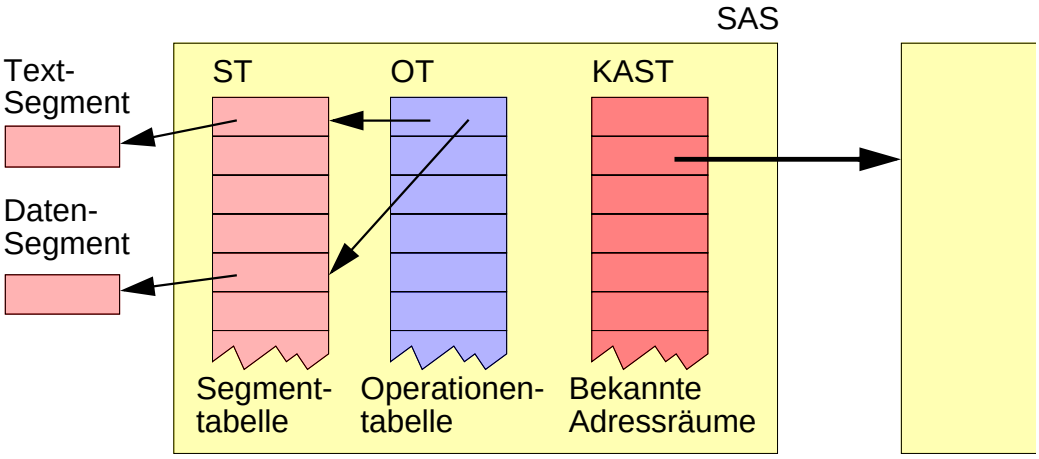
- Schutz vor gegenseitigem Speicherzugriff
  - Segmentierung und Seitenadressierung erlauben es, jedem Prozess nur den benötigten Speicher einzublenden
  - Segmentverletzung löst Unterbrechung aus
- Systemaufrufe
  - definierter Weg von einer Schutzumgebung (der des Prozesses) in eine andere (der des Betriebssystems)
- Erweiterung dieses Konzepts:
  - allgemeine Prozeduraufrufe zwischen verschiedenen Schutzumgebungen, realisiert mit der Speicherverwaltung und deren Hardware (MMU)

3.1 Modulkonzept von Habermann

- Idee (von 1976)
  - Adressräume (Module) bilden Schutzumgebungen
  - Adressräume bieten definierte Operationen an (ähnlich wie das Betriebssystem Systemaufrufe anbietet)
  - Parameter werden in speziellen Segmenten übergeben
- ★ Bietet allgemeinen Schutz der Module und erlaubt kontrollierte Interaktionen
- Module besitzen einen statischen Adressraum *(SAS, Static address space)*
  - enthält Liste von Segmenten, die zu dem Modul gehören bzw. von dem Modul zugegriffen werden dürfen
  - enthält Liste von angebotenen Operationen mit den Angaben, welche Segmente jede Operation benötigt (u.a. Segment für die auszuführenden Instruktionen)

3.1 Modulkonzept von Habermann (2)

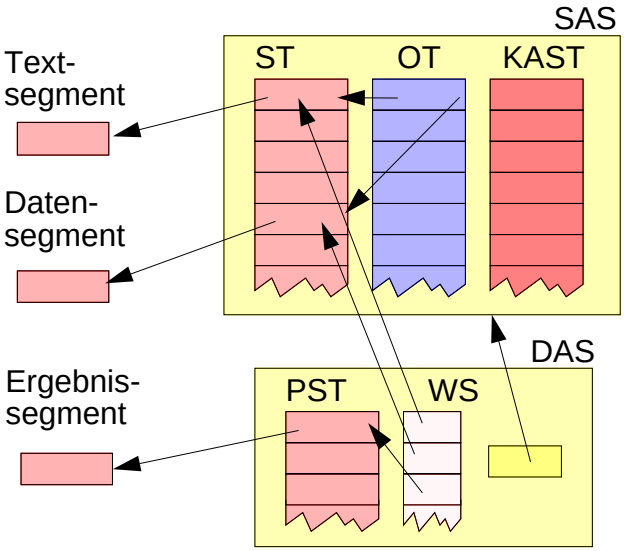
- enthält Liste von bekannten Adressräumen anderer Module (dort können dann Operationen aufgerufen werden)



KAST = *Known address space table*

3.1 Modulkonzept nach Habermann (3)

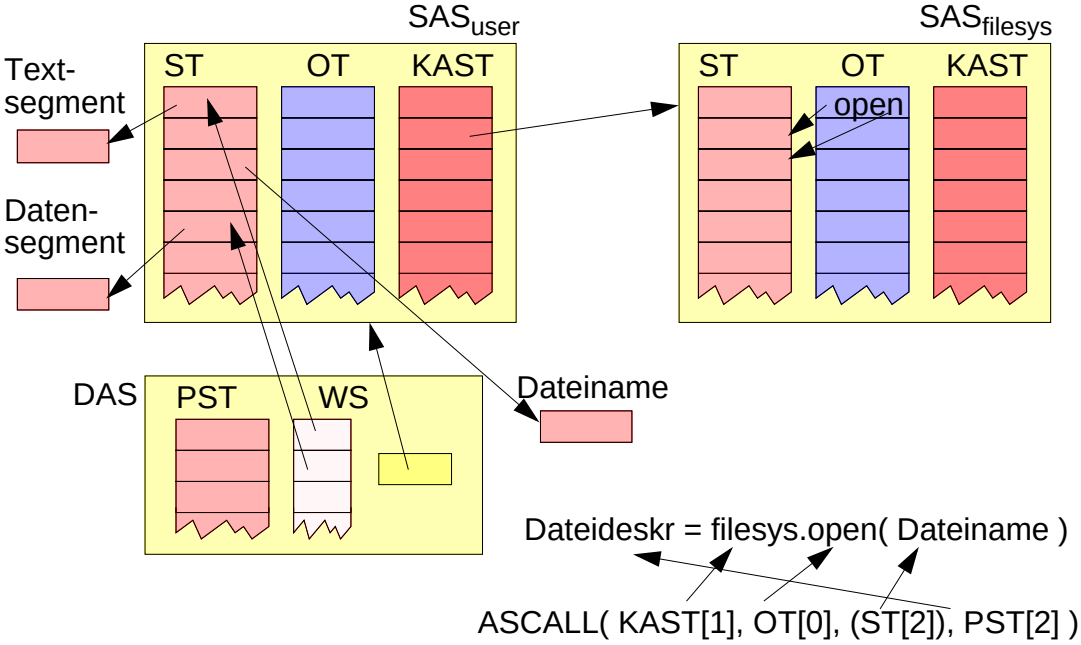
- Aktivitätsträger sind einen dynamische Adressraum zugeordnet *(DAS, Dynamic address space)*



- enthält Segmenttabelle für Parameter und Ergebnisse *(PST)*
- enthält Verweise auf Segmenttabellen, die den Adressraum zusammenstellen *(WS= Working space)*

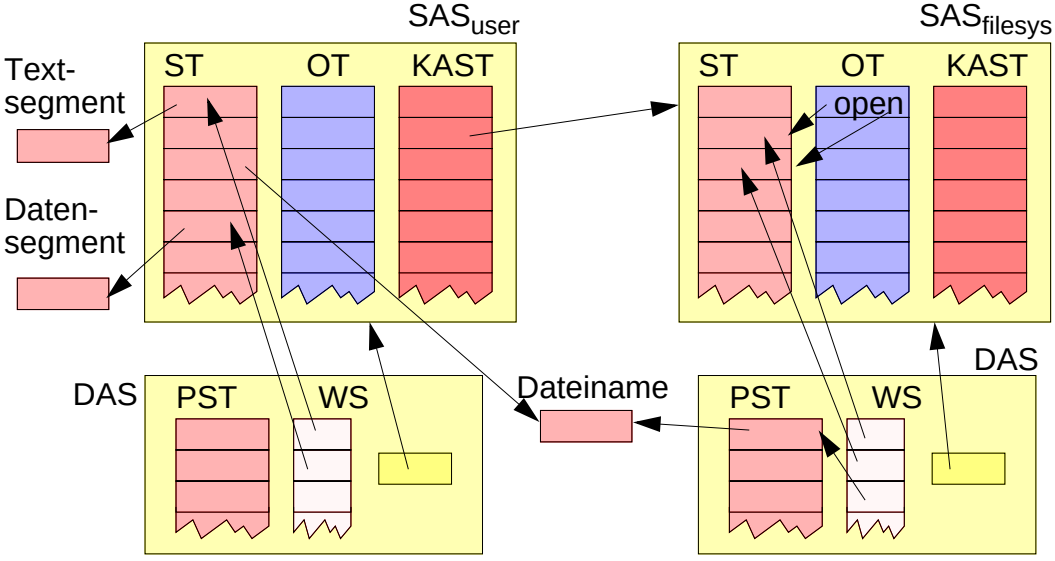
3.2 Beispielaufruf

- SAS des Benutzers ruft Operation „open“ des Dateisystem-SAS auf



3.2 Beispielaufruf (2)

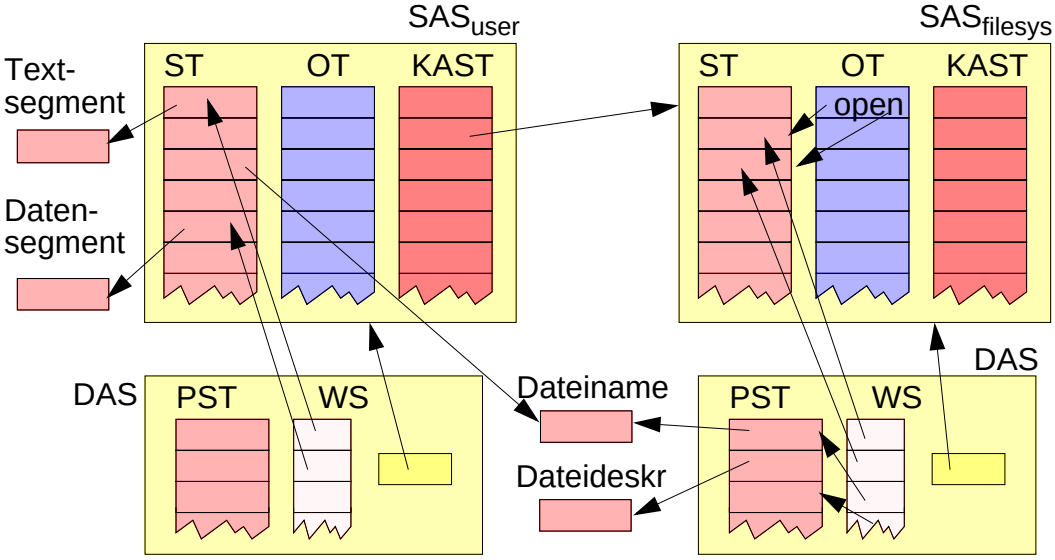
- SAS des Benutzers ruft Operation „open“ des Dateisystem-SAS auf



- für den Aufruf von „open“ wird ein neuer DAS erzeugt

3.2 Beispielaufruf (3)

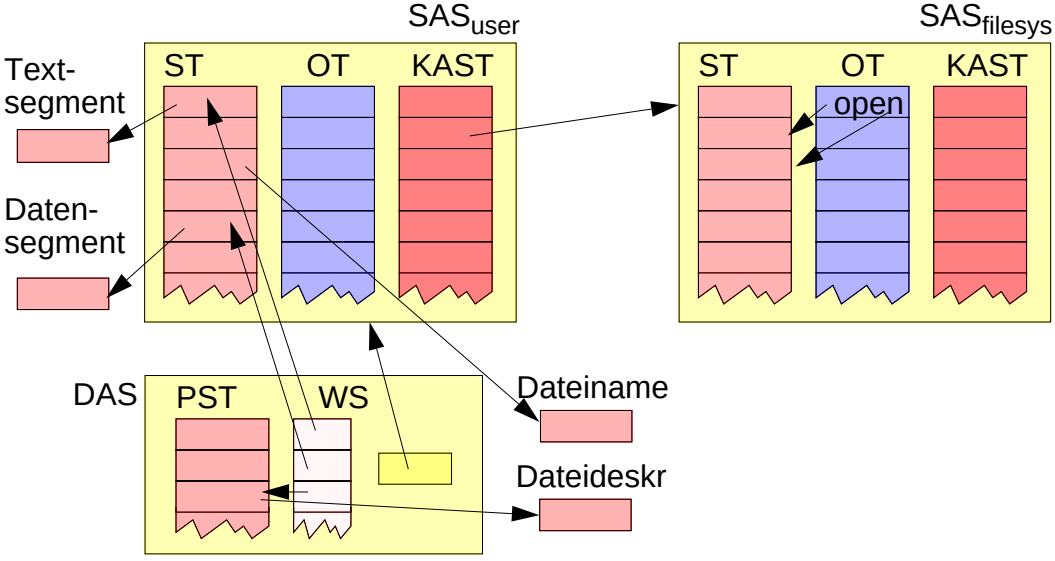
- SAS des Benutzers ruft Operation „open“ des Dateisystem-SAS auf



◆ „open“ erzeugt neues Segment für den Dateideskriptor

3.2 Beispielaufruf (4)

- SAS des Benutzers ruft Operation „open“ des Dateisystem-SAS auf



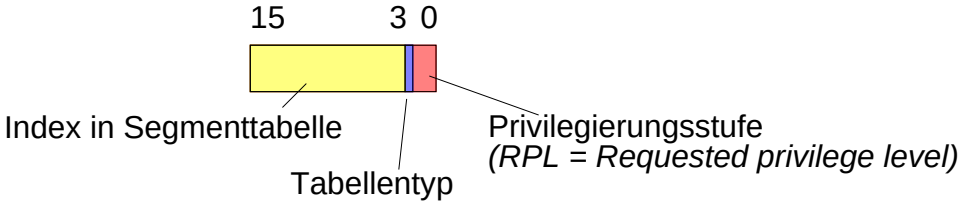
◆ Ergebnissegment wird an den Aufrufer zurückgegeben

3.3 Beispiel: Pentium

- Privilegierungsstufen
  - Stufe 0: höchste Privilegien (privilegierte Befehle, etc.): BS Kern
  - Stufe 1: BS Treiber
  - Stufe 2: BS Erweiterungen
  - Stufe 3: Benutzerprogramme
- Merke:
  - kleine Stufennummer: hohe Privilegien
  - große Stufennummer: kleine Privilegien

3.3 Beispiel: Pentium (2)

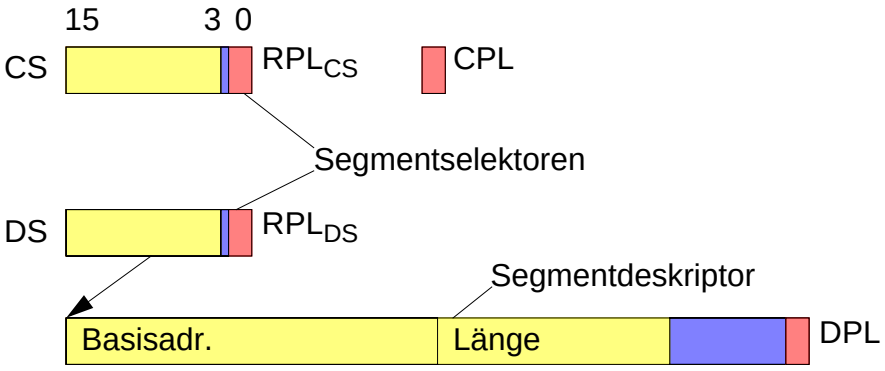
- Segmentselektoren enthalten Privilegierungsstufe



- Codesegmentselektor (CS) bestimmt aktuelle Privilegierungsstufe
  - CPL = *Current privilege level*

3.3 Beispiel: Pentium (3)

- Datenzugriff (z.B. auf Datensegment DS)



  - DPL = *Descriptor privilege level*
  - CPL ist normalerweise gleich RPL$_{CS}$
  - Zugriff wird erlaubt, wenn: DPL ≥ max(CPL, RPL$_{DS}$)
  - ansonsten wird Unterbrechung ausgelöst (Schutzverletzung)

3.3 Beispiel: Pentium (4)

- Erläuterung:
  - DPL < CPL: *Schutzverletzung*
    augenblickliche Privilegierungsstufe hat weniger Privilegien als im Deskriptor verlangt
    (CPL hat höhere Stufennr. als der Deskriptor)
  - DPL ≥ CPL: OK
    augenblickliche Privilegierungsstufe ist mindestens so hoch wie die im Deskriptor

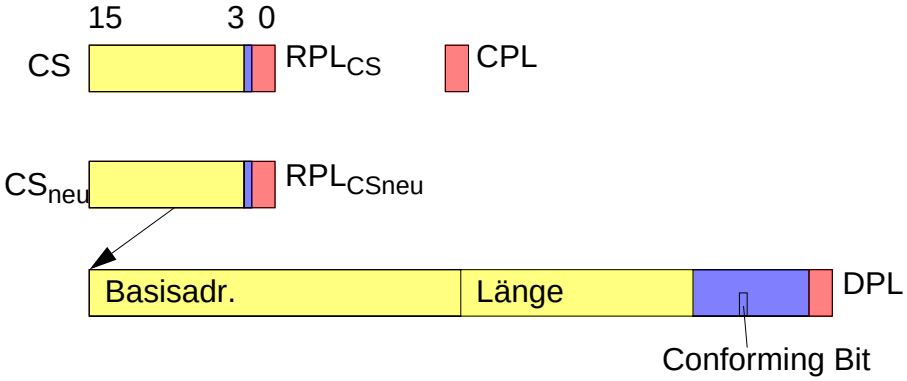
★ Segment kann nur angesprochen werden, wenn augenblickliche Stufe die gleichen oder mehr Privilegien beinhaltet.

3.3 Beispiel: Pentium (5)

- Erläuterung:
  - DPL < RPL(ds): *Schutzverletzung*
    Selektor hat weniger Privilegien als der Deskriptor
    (Selektor hat größere Stufe als der Deskriptor)
  - DPL ≥ RPL(ds): OK
    Selektor hat mindestens die gleiche Privilegierungsstufe wie der Deskriptor
- ★ Selektor darf keine geringeren Privilegien versprechen als wirklich verlangt sind.

3.3 Beispiel: Pentium (5)

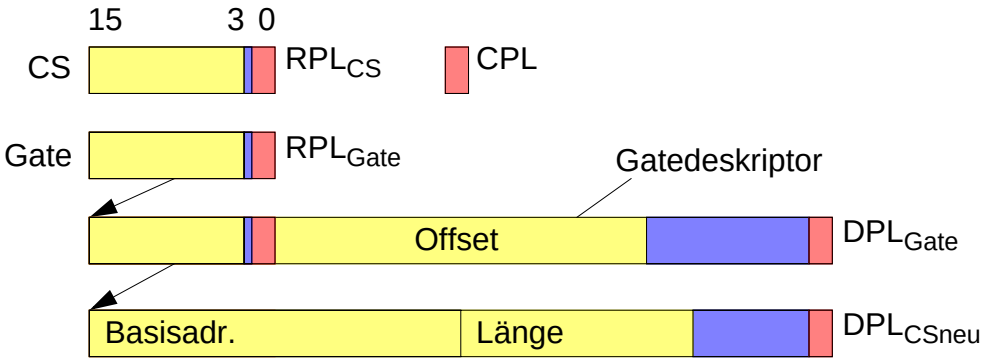
- Sprünge in andere Codesegmente *(Far Call)*



  - Sprung wird erlaubt, falls: DPL = CPL oder Conforming Bit gesetzt und DPL ≤ CPL
  - Im Falle von DPL ≤ CPL wird jedoch CPL nicht geändert (Codesegment hat höheres Privileg, CPL bleibt aber unverändert)

3.3 Beispiel: Pentium (6)

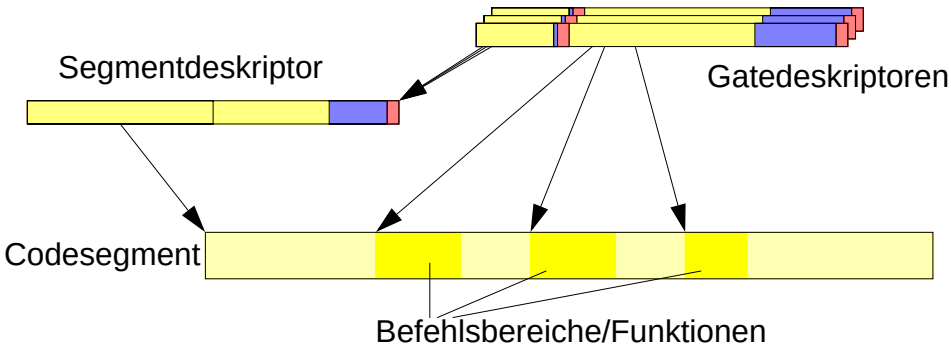
- Kontrolltransfer mit einem *Gate*



- Gatedeskriptoren stehen wie Segmentdeskriptoren in der Segmenttabelle
- Gatedeskriptor enthält Segmentselektor für das Codesegment und einen Offset zu diesem Segment, an dem der Einsprungpunkt liegt
- Kontrolltransfer (*CALL* Aufruf) wird erlaubt, falls:
  $DPL_{Gate} \geq \max(CPL, RPL_{Gate})$ und $DPL_{CSneu} \leq CPL$

3.3 Beispiel: Pentium (7)

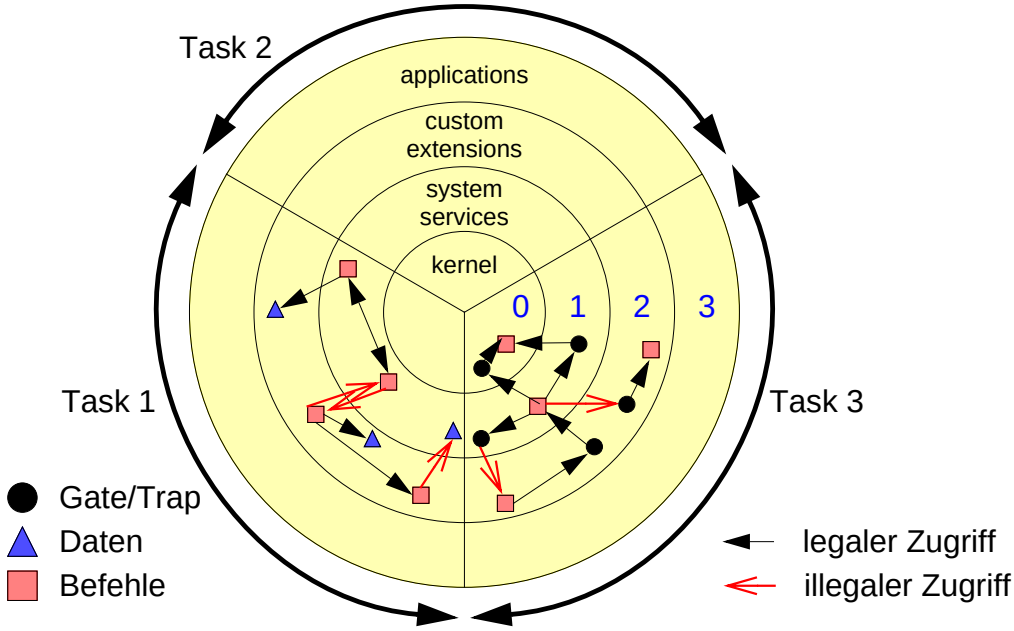
- Gates erlauben den kontrollierten Sprung in ein privilegierten Befehlsbereich



- es existiert auch der entsprechende Rücksprung
- für jede Privilegierungsstufe gibt es einen eigenen Stack; dieser wird mit umgeschaltet
- Parameter werden automatisch auf den neuen Stack kopiert (Anzahl wird im Gatedeskriptor vermerkt)

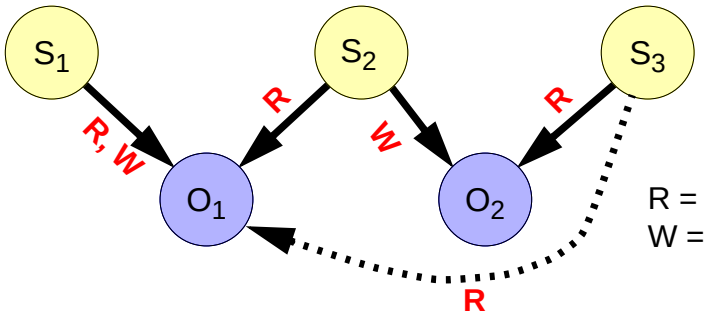
3.3 Beispiel: Pentium (8)

- Beispiel eines realisierten Schutzsystems



4 Capability-basierte Systeme

- Ein Benutzer (Subjekt) erhält eine Referenz auf ein Objekt
  - die Referenz enthält alle Rechte, die das Subjekt an dem Objekt besitzt
  - bei der Nutzung der Capability (Zugriff auf das Objekt) werden die Rechte überprüft



Subjekte und Objekte; Weitergabe einer Capability (O_1 von S_2 nach S_3)

4 Capability-basierte Systeme (2)

★ Vorteile

- ◆ keine Speicherung von Rechten beim Objekt oder Subjekt nötig; Capability enthält Zugriffsrechte
- ◆ leichte Vergabe von individuellen Rechten
- ◆ einfache Weitergabe von Zugriffsrechten möglich

▲ Nachteile

- ◆ Weitergabe nicht kontrollierbar
- ◆ Rückruf von Zugriffsrechten nicht möglich
- ◆ Capability muss vor Fälschung und Verfälschung geschützt werden (z.B. durch kryptographische Mittel oder durch Speicherverwaltung)

4.1 Beispiel: Hydra

■ Hydra ist ein Capability-basiertes Betriebssystem

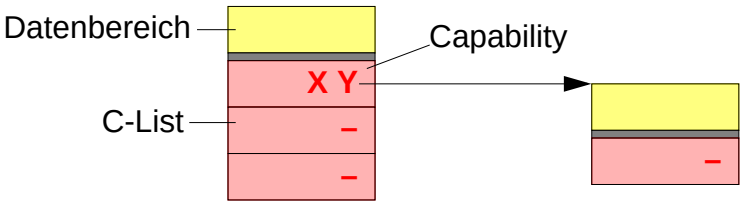
- ◆ entwickelt Mitte der Siebziger Jahre an der Carnegie-Mellon University
- ◆ lief auf einem speziellen Multiprozessor namens **C.mmp**
- ◆ Capability-Mechanismen sind integraler Bestandteil des Betriebssystems

■ Objekte in Hydra werden durch Capabilities angesprochen und geschützt

- ◆ Objekte haben einen Typ (z.B. Prozeduren, Prozesse/LNS, Semaphoren, Datei etc.)
- ◆ Capabilities haben entsprechenden Typ
- ◆ benutzerdefinierte Typen sind möglich

4.1 Beispiel: Hydra (2)

- generische Operationen für alle Typen implementiert durch das Betriebssystem
- Objekte besitzen eine Liste von Capabilities auf andere Objekte (genannt *C-List*)
- Capabilities enthalten Rechte
- Objekte besitzen einen Datenbereich (implementiert durch geschütztes Segment)



4.1 Beispiel: Hydra (3)

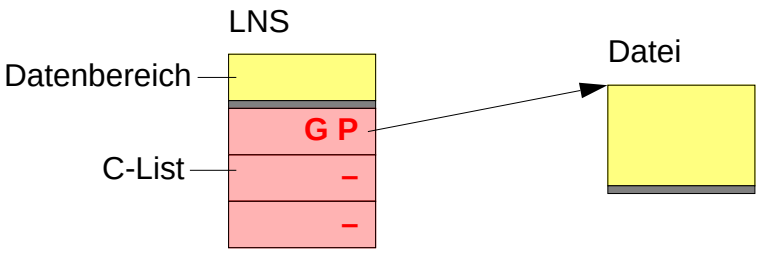
- Prozesse (Subjekte)
  - Prozesse besitzen einen aktuellen Kontext, den LNS *(Local name space)*
  - LNS ist ein Objekt
  - zum LNS gehört einen Aktivitätsträger (Thread)
  - LNS kann nur auf Objekte zugreifen, die in seiner C-List stehen (mehrstufige Zugriffe, z.B. auf die C-List eines Objekts, dessen Capabilities in der C-List des LNS steht, sind möglich; Pfad zur eigentlichen Capability)
- Capabilities
  - Prozesse können nur über Systemaufrufe ihre Capabilities bzw. ihre C-List bearbeiten
  - Capabilities können nicht gefälscht oder verfälscht werden
  - Betriebssystem kann sicheres Schutzkonzept basierend auf Capabilities implementieren

4.2 Datenzugriff in Hydra

- Operationen auf dem Datenbereich
  - *Getdata:* kopiere Abschnitt aus dem Datenbereich eines Objekts in den Datenbereich des LNS
  - *Putdata:* kopiere Abschnitt aus dem Datenbereich des LNS in den Datenbereich eines Objekts
  - *Adddata:* füge Daten zu dem Datenbereich eines Objekts hinzu
- Dazugehörige Rechte:
  - ***G****ETRTS:* erlaubt den Aufruf von *Getdata*
  - ***P****UTRTS:* erlaubt den Aufruf von *Putdata*
  - ***AD****DRTS:* erlaubt den Aufruf von *Adddata*
- Rechte müssen in der Capability zum Objekt gesetzt sein

4.2 Datenzugriff in Hydra (2)

- Beispiel: Implementierung von Dateien
  - *Getdata* erlaubt das Lesen von Daten
  - *Putdata* erlaubt das Schreiben von Daten
  - *Adddata* erlaubt das Anhängen von Daten
- Entsprechende Rechte können pro Capability gesetzt werden

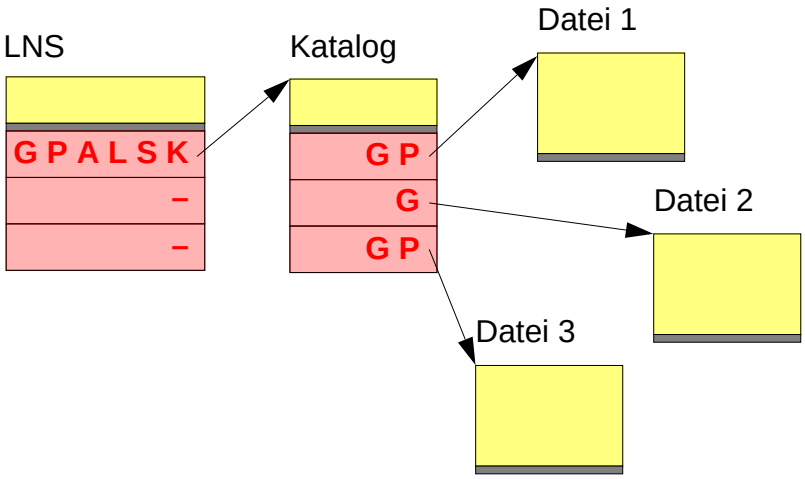


4.3 Zugriff auf Capabilities in Hydra

- Operationen auf der C-List:
  - *Load:* kopieren einer Capability aus der C-List eines Objekts in die C-List des LNS
  - *Store:* kopieren einer Capability aus der C-List des LNS in die C-List eines Objekts (dabei können Rechte maskiert werden)
  - *Append:* anfügen einer Capability in die C-List eines Objekts
  - *Delete:* löschen einer Capability aus der C-List eines Objekts
- Rechte:
  - ***L****OADRTS:* erlaubt Aufruf von *Load*
  - ***S****TORTS:* erlaubt Aufruf von *Store*
  - ***A****PPRTS:* erlaubt Aufruf von *Append*
  - ***K****ILLRTS:* erlaubt Aufruf von *Delete*

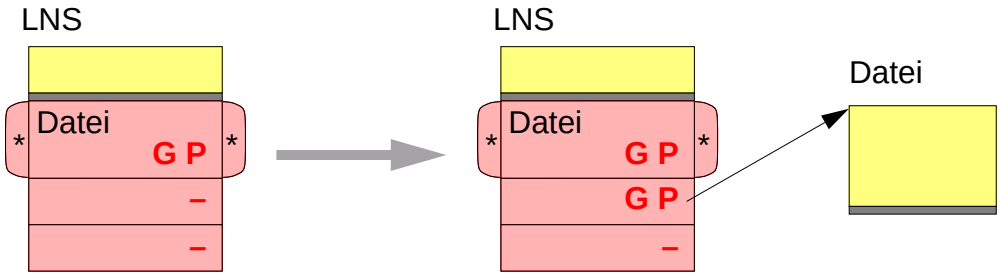
4.3 Zugriff auf Capabilities in Hydra (2)

- Beispiel: Implementierung von Katalogen
  - *Load* erlaubt das Auflösen von Namen (Aufrufer bekommt die Capability)
  - *Store* und *Append* erlauben das Hinzufügen von Dateien zum Katalog
  - *Delete* erlaubt das Austragen von Dateien aus dem Katalog



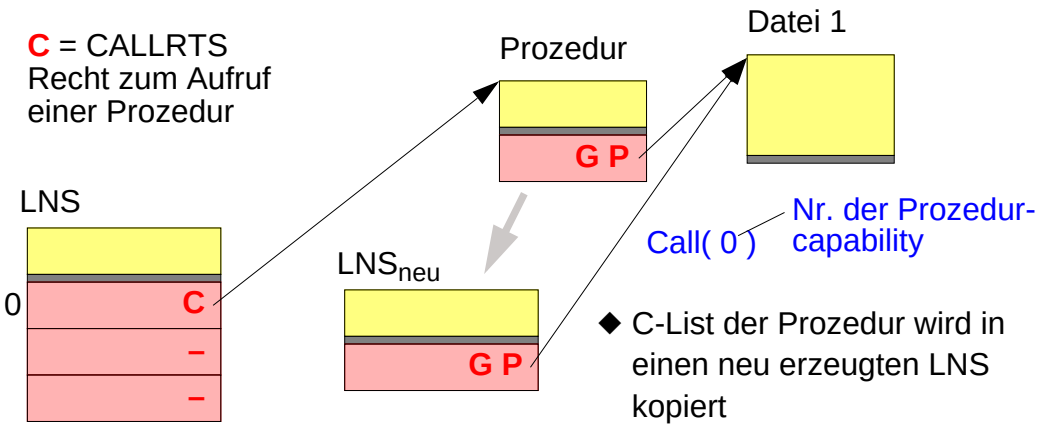
4.4 Objekterzeugung in Hydra

- Objekterzeugung über Erzeugungsschablonen *(Creation Templates)*
  - Erzeugungsschablone enthält den Typ des neu zu erzeugenden Objektes und eine Rechtemaske
  - nur die in der Maske angeschalteten Rechte werden dem Aufrufer in einer neuen Capability gegeben



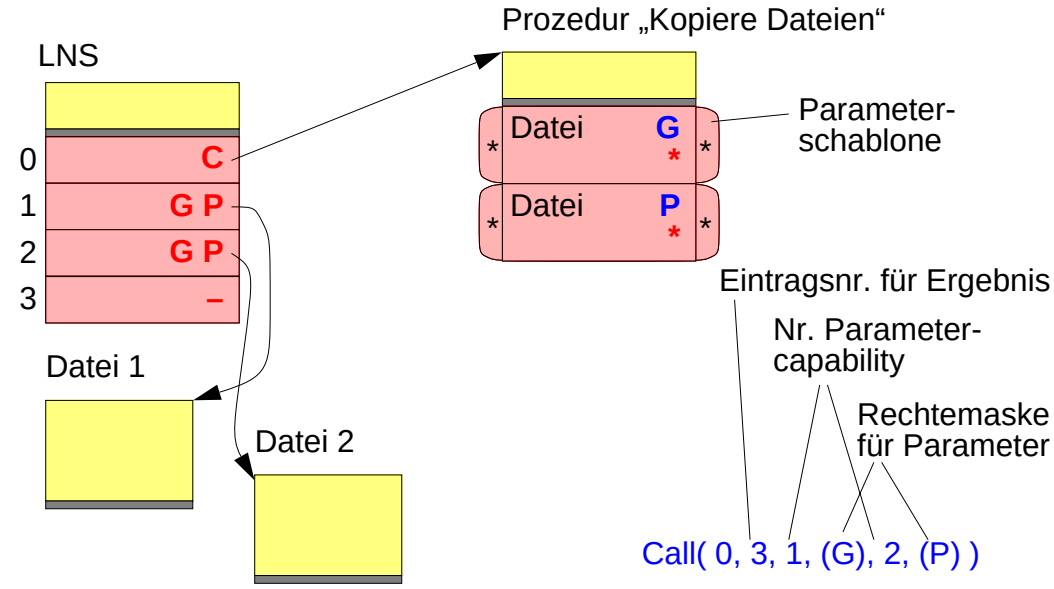
4.5 Prozeduraufruf in Hydra

- Prozedur ist ein Objekt, aus dem beim Aufruf ein LNS des laufenden Prozesses erzeugt wird
  - neuer LNS wird aktueller Kontext (alte LNS stehen auf einem Stack; sie werden wieder aktiviert, wenn Prozedur zu Ende)
- Aufruf einer Prozedur



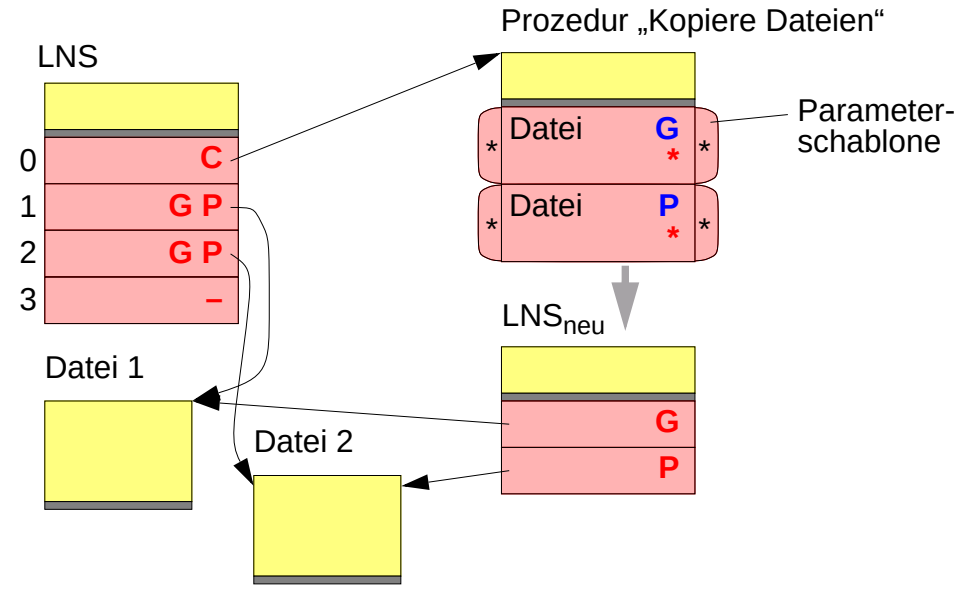
4.5 Prozeduraufruf in Hydra (2)

- Übergabe von Parametern
  - Beispiel: Prozedur zum Kopieren von Dateiinhalten



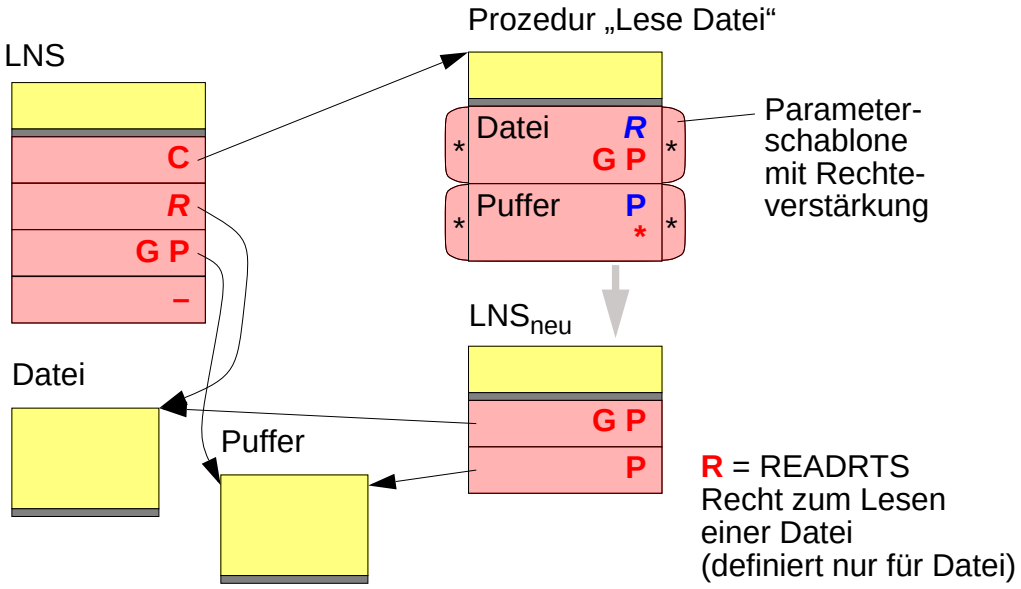
4.5 Prozeduraufruf in Hydra (3)

- Übergabe von Parametern
  - Beispiel: Prozedur zum Kopieren von Dateiinhalten



4.5 Prozeduraufruf in Hydra (3)

- Verstärken von Rechten
  - Beispiel: Prozedur zum Lesen von Dateiinhalten in einen Puffer



4.6 Problem: Gegenseitiges Misstrauen

- Aufrufer misstraut einer Prozedur
  - Aufrufer möchte der Prozedur nur soviel Rechte einräumen wie nötig
- Aufgerufene Prozedur misstraut dem Aufrufer
  - Aufrufer soll nur soviel Rechte und Zugang bekommen wie erforderlich
- ★ Hydra Prozeduraufruf unterstützt diese Forderungen direkt
  - Aufrufer übergibt Capabilities, die nötig sind
  - Aufrufer kann Rechte bei der Übergabe maskieren und damit ausschalten
  - Aufrufer erhält nur Zugang zu einem definierten Ergebnis
  - Prozedur kann eigene Capabilities besitzen, die einem LNS zur Verfügung stehen und die dem Aufrufer verborgen bleiben können

4.6 Problem: Gegenseitiges Misstrauen (2)

▲ Rechteverstärkung als Sicherheitslücke?

- ◆ Verstärkungsschablone wird nur an vertrauenswürdige Prozeduren ausgegeben und kann nicht einfach erzeugt werden

4.7 Problem: Modifikationen

■ Aufrufer möchte Modifikationen an und über Parameter ausschließen

- ◆ eine Prozedur soll nichts verändern können

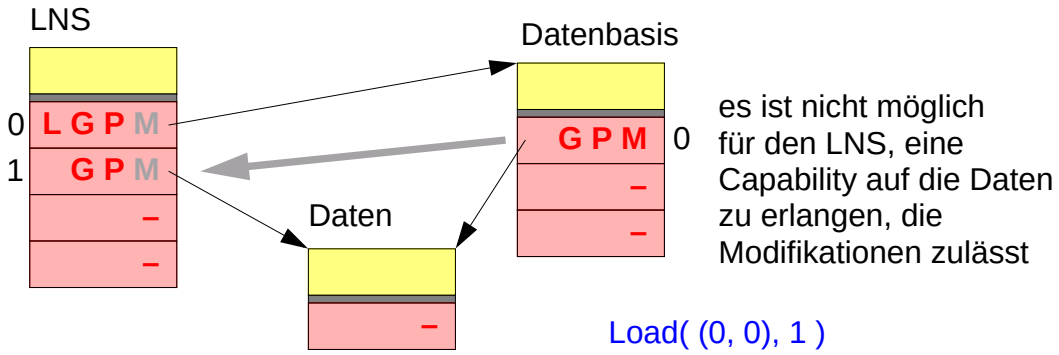
■ Wegnehmen der entsprechenden Rechte langt nicht

- ◆ Prozedur kann lesend zu neuen Capabilities gelangen und über diese Änderungen vornehmen (Transitivität)
- ◆ Rechteverstärkung könnte angewandt werden

4.7 Problem: Modifikation (2)

★ Einführung des Modifikationsrechts ***M***DFYRTS

- ◆ für alle modifizierenden Operationen an Datenbereichen und C-Lists muss zusätzlich das Modifikationsrecht für das Objekt vorhanden sein
- ◆ Modifikationsrecht wird automatisch gelöscht, wenn eine Capability über einen Pfad geladen wird, auf dem eine der Capabilities kein Modifikationsrecht besitzt
- ◆ Modifikationsrecht kann nicht über Rechteverstärkung erlangt werden



4.7 Problem: Modifikation (3)

- Parameterübergabe
  - Wegnahme des Modifikationsrecht bei Parametern stellt sicher, dass die aufgerufene Prozedur keinerlei Veränderungen beim Aufrufer durchführen kann

4.8 Problem: Ausbreitung von Capabilities

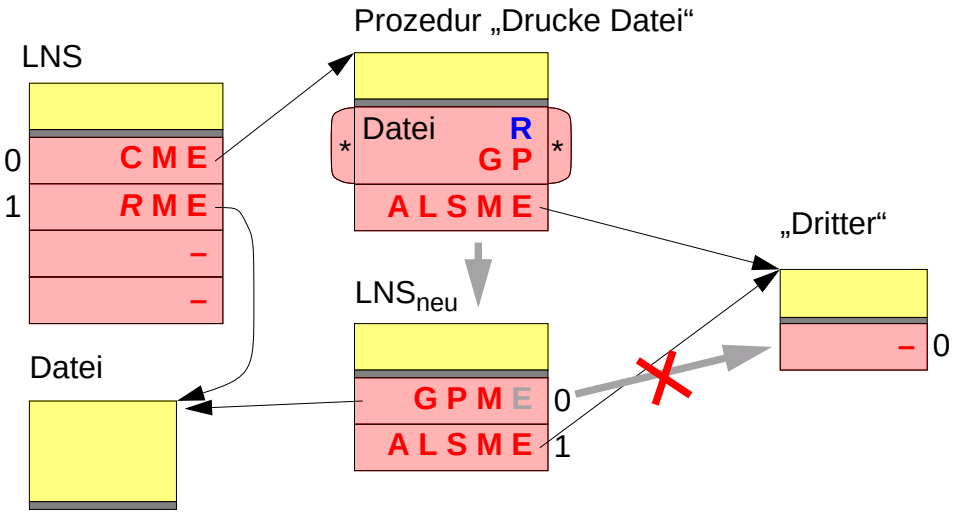
- Aufrufer will verhindern, dass eine übergebene Capability vom Aufgerufenen an einen Dritten weitergegeben wird *(Propagation Problem)*
  - Beispiel: Prozedur „Drucken“ soll niemandem eine Referenz auf die zu druckenden Daten weitergeben können

4.8 Problem: Ausbreitung von Capabilities (2)

- ★ Einführung des Environment-Rechts ***E****NVRTS*
  - für das Speichern oder Anfügen einer Capability an eine C-List muss die zu speichernde Capability selbst das Environment-Recht besitzen
  - Environment-Recht wird automatisch gelöscht, wenn eine Capability über einen Pfad geladen wird, auf dem eine der Capabilities kein Environment-Recht besitzt
  - Environment-Recht kann nicht über Rechteverstärkung erlangt werden

4.8 Problem: Ausbreitung von Capabilities (3)

- Versuchte Weitergabe einer Capability an einen Dritten



Call(0, 2, 1, (–E)) Store(0, (1, 0))

4.9 Problem: Aufbewahrung von Capabilities

- Aufrufer möchte sicher sein, dass Aufgerufener keine Capabilities nach der Bearbeitung des Aufrufs zurückbehalten kann *(Conservation Problem)*

★ Environment-Recht zusammen mit dem Aufrufmechanismus genügt

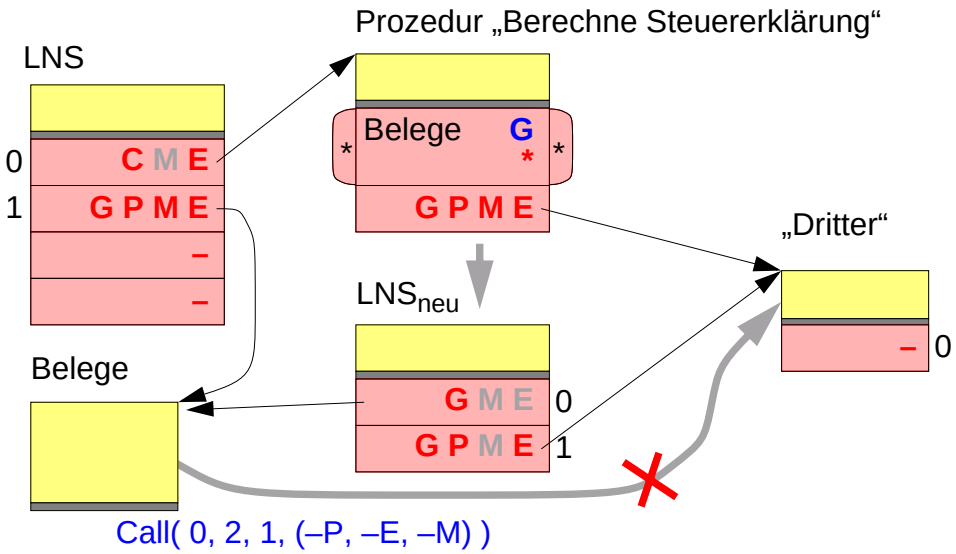
- ◆ Aufgerufener kann Capability ohne ENVRTS nicht weitergeben und folglich nicht abspeichern
- ◆ der LNS des Aufrufs wird mit Beendigung des Aufrufs vernichtet, so dass die übergebenen Capabilities nicht zurückbehalten werden können
- ◆ ENVRTS wirkt transitiv, so dass auch die über eine Parameter-Capability gewonnenen Capabilities nicht weitergegeben werden können

4.10 Problem: Informationsflussbegrenzung

- Aufrufer möchte die Verbreitung von Informationen aus übergebenen Parametern einschränken *(Confinement Problem)*
  - selektiv: bestimmte Informationen sollen nicht nach außen gelangen
  - global: gar keine Informationen sollen nach außen gelangen
  - ENVRTS ist nicht ausreichend, da Prozedur den Dateninhalt von Parameterobjekten kopieren könnte
    (ENVRTS wirkt nur auf die Weitergabe von Capabilities)
- Hydra realisiert nur globale Informationsflussbegrenzung
- ★ Modifikationsrecht auf der Prozedur-Capability
  - wenn kein Modifikationsrecht vorhanden ist, werden bei allen in den LNS übernommenen Capabilities die Modifikationsrechte ausgeschaltet (gilt jedoch nicht für Parameter)

4.10 Problem: Informationsflussbegrenzung (2)

- Beispiel: Prozedur zur Steuerberechnung
  - die übergebenen Beleg- und Buchhaltungsdaten sollen nicht weitergegeben werden können



4.11 Problem: Initialisierung

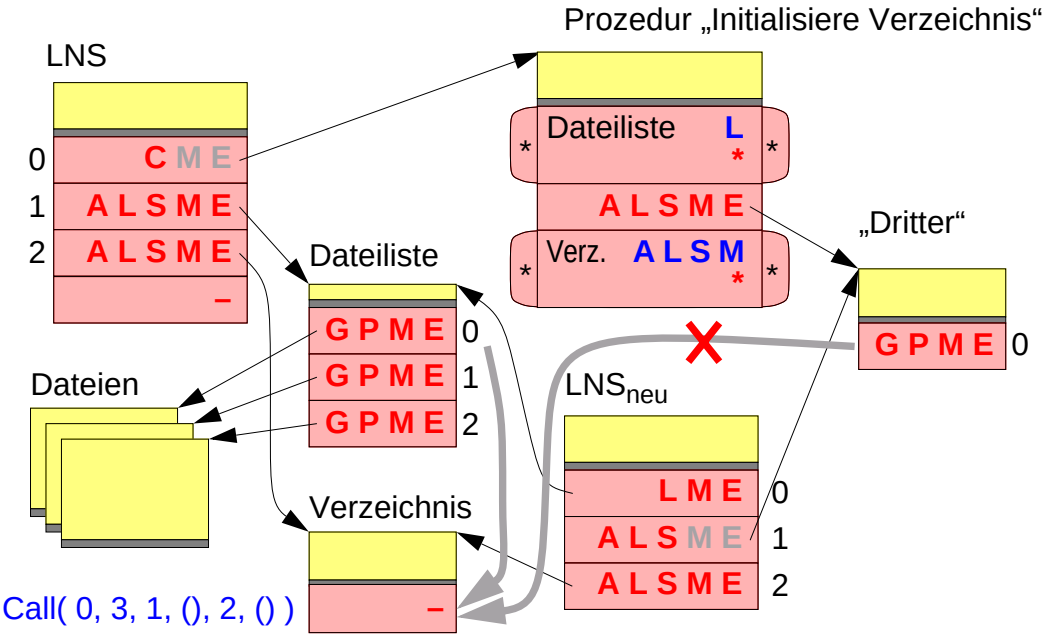
- Initialisierung von Objekten durch Prozeduren
  - Übergabe eines Objekts und verschiedener Capabilities, mit denen das Objekt initialisiert werden soll
  - Problem: Parameter-Capabilities müssen Environment-Recht besitzen (sonst ist das zu initialisierende Objekt nicht arbeitsfähig), gleichzeitig soll aber die Ausbreitung solcher Capabilities eingeschränkt werden
    - Lösung: Wegnahme des Modifikationsrechts auf der Prozedurcapability
  - Problem: Es muss verhindert werden, dass die Prozedur in das zu initialisierende Objekt eigene oder fremde Capabilities einsetzt, so dass es später Einfluss auf das zu initialisierende Objekt nehmen kann
- Beispiel: Prozedur zur Initialisierung eines Katalogs bekommt Capabilities auf die entsprechenden Dateien
  - es soll sichergestellt werden, dass Prozedur keine eigenen Dateicapabilities in den Katalog einfügt

4.11 Problem Initialisierung (2)

- ★ Environment-Recht auf der Prozedur-Capability
  - wenn kein Environment-Recht vorhanden ist, werden bei allen in den LNS übernommenen Capabilities die Environment-Rechte ausgeschaltet (gilt jedoch nicht für Parameter)
  - durch das fehlende Environment-Recht können alle bereits vorhandenen Capabilities nicht in das zu initialisierende Objekt gespeichert werden

4.11 Problem: Initialisierung (3)

- Beispiel: Initialisierung eines Verzeichnis

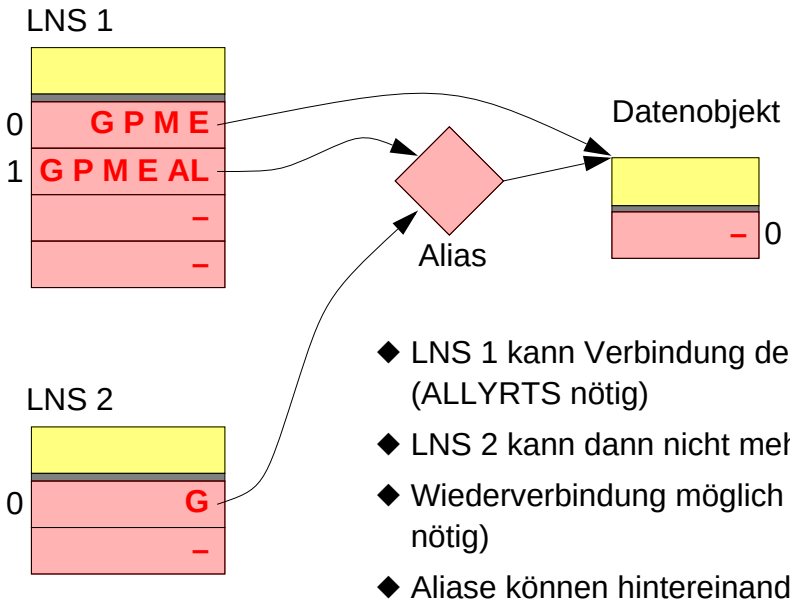


4.12 Rückruf von Capabilities

- Anwender möchte ausgegebene Capabilities für ungültig erklären
  - sofortiger Rückruf — Rückruf nach einiger Zeit erst wirksam
  - dauerhafter Rückruf — Rückruf nur zeitlich begrenzt wirksam
  - selektiver Rückruf — Rückruf für alle Benutzer eines Objekts
  - partieller Rückruf — Rückruf aller Rechte an einem Objekt
  - Recht zum Rückruf; Rückruf des Rückrufrechts
- ★ Hydra setzt sogenannte Aliase ein
  - Alias ist eine Indirektionsstufe zu Capabilities
  - Statt auf ein Objekt können Capabilities auf Aliase verweisen und diese wiederum auf andere Aliase oder schließlich auf das eigentliche Objekt
  - Verbindung vom Alias zum Objekt kann gelöst werden: Fehler beim Zugriff
  - Recht zum Lösen der Verbindung ***AL****LYRTS*
    (engl. *ally* = verbünden)

4.12 Rückruf von Capabilities (2)

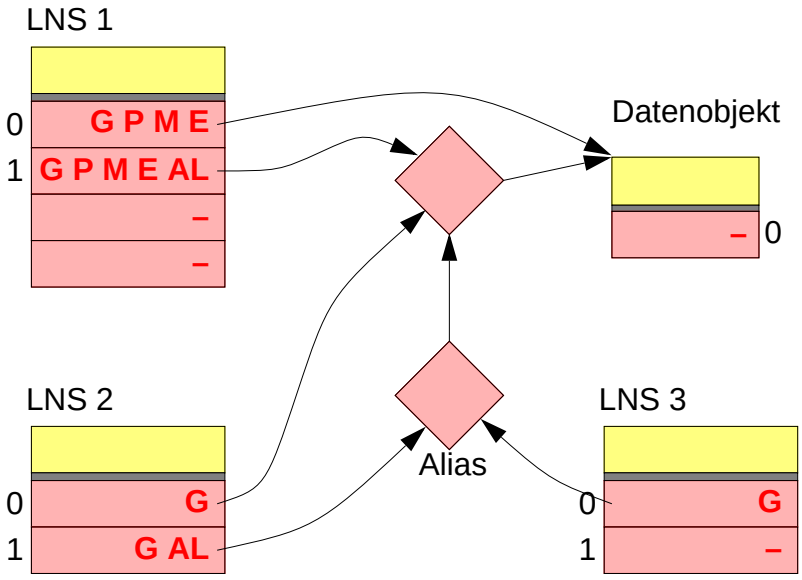
- Beispiel: Weitergabe einer rückrufbaren Capability



- LNS 1 kann Verbindung des Alias lösen (ALLYRTS nötig)
- LNS 2 kann dann nicht mehr zugreifen
- Wiederverbindung möglich (Originalcapability nötig)
- Aliase können hintereinander auftreten

4.12 Rückruf von Capabilities (3)

- Beispiel: Aliasketten

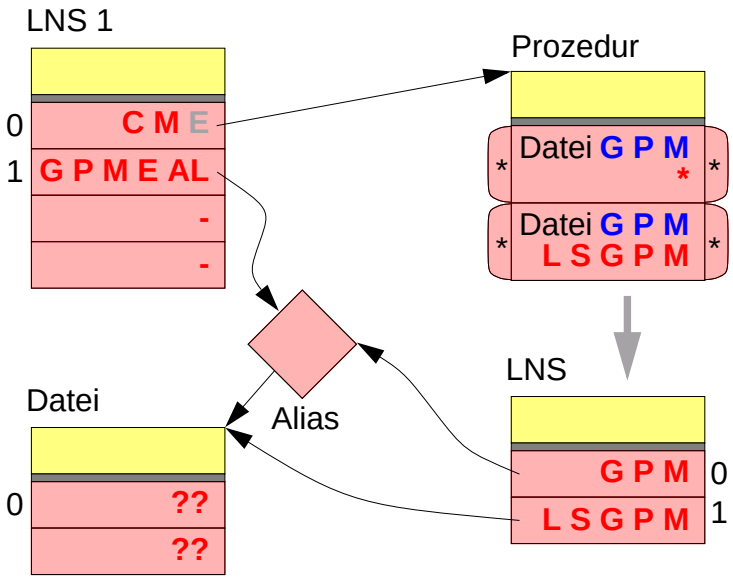


4.12 Rückruf von Capabilities (4)

- ▲ Problem: Rückruf während der Bearbeitung eines Objekts
  - ◆ inkonsistente Zustände möglich
- ★ Lösung in Hydra
  - ◆ Parameter-Capabilities, die durch eine rechteverstärkende Parameterschablone angenommen werden, zeigen auf das Originalobjekt
- ▲ Nachteil
  - ◆ nicht vertrauenswürdige Prozeduren können rückruffreie Capability erlangen
  - ◆ Problem fällt in die selbe Kategorie wie rechteverstärkende Parameterschablonen an sich

4.12 Rückruf von Capabilities (5)

- ■ Beispiel:



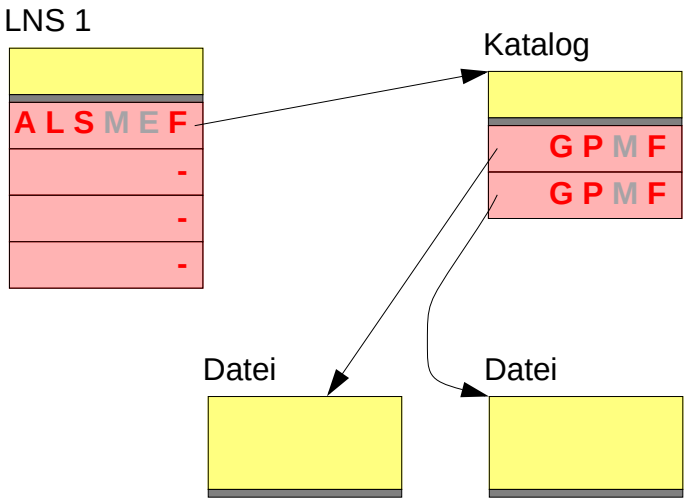
Call(0, 2, 1, (–E), 1, (–E))

4.13 Garantierter Zugriff

- Schutz vor Rückruf
  - ◆ Modifizierende Benutzer eines Objekts
    - kooperierende Benutzer: Rückruf keine Gefahr
    - nicht kooperierende Benutzer: Rückruf nötig
  - ◆ Benutzer eines Objekts ohne modifizierende Zugriffe
    - Aufruf von Prozeduren ist unkritisch, da Aufrufe nicht rückrufbar sind
    - lesende Zugriffe auf Objekte sind kritisch:
      Verhindern des Rückrufs ist jedoch nicht ausreichend
      – Daten im Objekt könnten gelöscht oder verfälscht werden
      – Capabilities im Objekt oder deren Rechte könnten entfernt werden
- ★ Hydra führt das Einfrierrecht *(Freeze right **F**RZRTS)* ein
  - ◆ Einfrieren nimmt Modifikationsrecht weg
  - ◆ ein Objekt kann nur gefroren werden, wenn alle Capabilities in der C-List bereits das Einfrierrecht haben

4.13 Garantierter Zugriff (2)

- Beispiel:



4.14 Bewertung von Hydra

- Hydra demonstrierte die Beherrschbarkeit einer ganzen Reihen von Sicherheitsproblemen
  - Ergebnisse flossen in eine ganze Reihe von Systemen
  - reine Capability-basierte Systeme haben sich jedoch nie durchgesetzt
- Hydras Probleme
  - lagen im wesentlichen nicht am Capability-Mechanismus
  - es gabe keine vernünftigen Editoren und Compiler
  - Hardware besaß keine Spezialhardware zur Unterstützung von Paging

5 Kryptographische Maßnahmen

- Verschlüsseln und Entschlüsseln vertraulicher Daten
  - aus den verschlüsselten Daten soll die Originalinformation nur mit Hilfe eines Schlüssels restauriert werden können
  - Schlüssel bleibt geheim
- Authentisierung
  - Empfänger kann verifizieren, wer der Absender ist
- Sicherer Kanal
  - gesendete Informationen können nicht gefälscht und verfälscht werden
  - nur der adressierte Empfänger kann die Informationen lesen
  - Empfänger kann den Absender authentisieren

5 Kryptographische Maßnahmen (2)

- Funktionen
  - Verschlüsselungsfunktion *E (encrypt): E(K_1, T)* → *C*
  - Entschlüsselungsfunktion *D (decrypt): D(K_2, C)* → *T*
  - *K* = Schlüssel, *T*= zu verschlüsselnder Text/Daten
- Verwandte Schlüssel
  - es gilt: K_1 und K_2 sind verwandt, wenn gilt: $\forall T: D(K_2, E(K_1, T)) = T$
- Symmetrisches Verschlüsselungsverfahren
  - es gilt: $K_1 = K_2$

5 Kryptographische Maßnahmen (3)

- Forderungen an ein Verschlüsselungsverfahren
  - Wenn K_2 unbekannt ist, soll es sehr aufwendig sein aus *E(K_1, T)* das *T* zu ermitteln (Entschlüsselungsangriff)
  - Es soll sehr aufwendig sein aus *T* und *E(K_1, T)* den Schlüssel K_1 zu ermitteln (Klartextangriff)
  - Bei asymmetrischen Verfahren soll es sehr aufwendig sein, aus K_1 den Schlüssel K_2 zu ermitteln und umgekehrt.

5.1 Monoalphabetische Verfahren

- Verfahren nach Caesar

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E

  - Verschlüsselungsfunktion: $E: M \rightarrow (M + k) \bmod 26$
  - k ist variierbar (26 Möglichkeiten)

- Zufällige Substitution

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
F	Q	H	A	J	U	L	G	N	S	P	W	R	O	T	C	V	Y	X	M	Z	K	B	I	D	E

  - 26! Möglichkeiten

5.1 Monoalphabetische Verfahren (2)

- ★ Nachteil
  - vollständiges Ausprobieren möglich bei Caesar
  - Häufigkeitsanalyse der Buchstaben
    - für eine Sprache gibt es häufigere Buchstaben, z.B. e im Deutschen
    - durch die Häufigkeitsanalyse können die Möglichkeiten stark eingeschränkt werden; vollständiges Probieren wird ermöglicht

5.2 Polyalphabetische Verschlüsselung

- Einsatz von vielen Abbildungen, die durch einen Schlüssel ausgewählt werden
  - Beispiel: Vigenére (Caesar-Verschlüsselung mit zyklisch wiederholten Folgen von Verschiebungswerten)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
...																										
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
X	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

5.2 Polyalphabetische Verschlüsselung (2)

- Auswahl der Zeile durch den entsprechenden Buchstaben des Schlüsselwortes

W	I	C	H	T	I	G	E	N	A	C	H	R	I	C	H	T
G	E	H	E	I	M	G	E	H	E	I	M	G	E	H	E	I

Originaltext

Schlüsselwort (wiederholt)

C	M	J	L	B	U	M	I	U	E	K	T	X	M	J	L	B

verschlüsselter Text

- Gilt als nicht sicher
  - Koinzidenzanalyse
  - Häufigkeitsanalysen und Brute force Attacke

5.2 Polyalphabetische Verfahren (3)

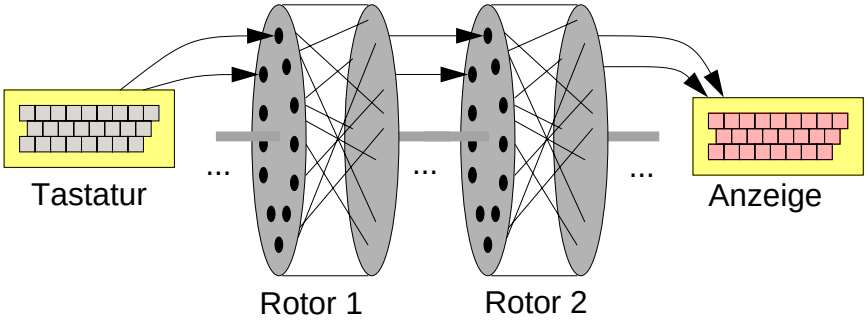
- Koinzidenz
  - Wahrscheinlichkeit für zwei gleiche Buchstaben untereinander bei umbrechendem Text
    - zufällige Buchstabenwahl: 3,8%
    - englischer Text: 6,6%
  - Brechen polyalphabetischer Verfahren
    - Bestimmen der Koinzidenz für verschiedene Textlängen
    - Textlänge mit höchster Koinzidenz ist wahrscheinlich Schlüsseltextlänge
    - danach Häufigkeitsanalyse pro Buchstabe des Schlüsseltexts

5.3 One-Time Pad Verfahren

- Theoretisch sicheres Verfahren
  - Liste von Zufallszahlen (soviele wie Zeichen in der Nachricht): *r[i]*
  - Zeichen *z[i]* der Nachricht wird verschlüsselt mit $c[i] = (z[i] + r[i]) \bmod 26$
  - Empfänger braucht die gleiche Liste
  - theoretisch sicher, da aus dem *c[]* nicht auf *z[]* geschlossen werden kann
- Praktisch unbrauchbar
  - echte Zufallszahlen nötig
  - lange Liste nötig
    - jede Liste kann nur einmal verwendet werden
    - Liste muss so lang wie die Nachricht sein

5.4 Rotormaschinen

- Drehende Scheiben verändern ständig die Permutation

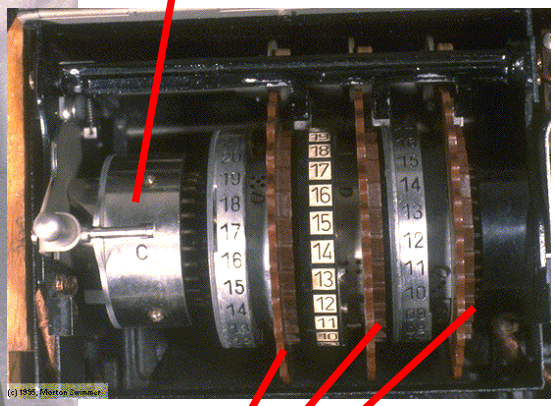


  - Einstellen einer Anfangsposition für die Rotoren
  - bei jedem Zeichen wird erster Rotor um eine Position weitergedreht
  - zweiter Rotor rotiert mit niedrigerer Geschwindigkeit
  - zum Entschlüsseln sind entsprechende Gegenstücke nötig

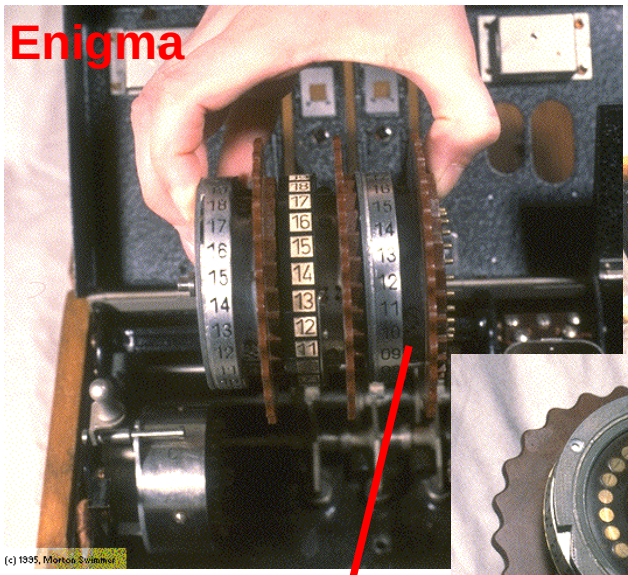
5.4 Rotormaschinen (2)

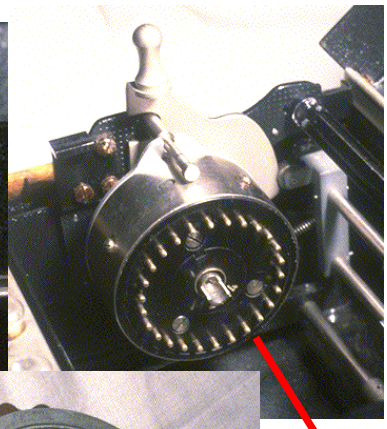
- Enigma
  - deutsche Chiffriermaschine aus dem zweiten Weltkrieg
  - drei Rotore und Reflektor
    - Reflektor leitet Strom wieder bei einer anderen Position durch die Rotoren zurück: Verfahren wird symmetrisch
    - Entschlüsseln mit den gleichen Rotoren möglich
- Verfahren gilt als nicht sicher
  - Brute force Attacke: *Collosus* Computer
- Schlüsseldemo
  - http://www.ugrad.cs.jhu.edu/~russell/classes/enigma/

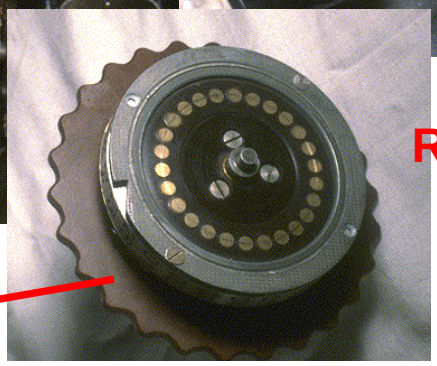
5.4 Rotormaschinen



5.4 Rotormaschinen (2)



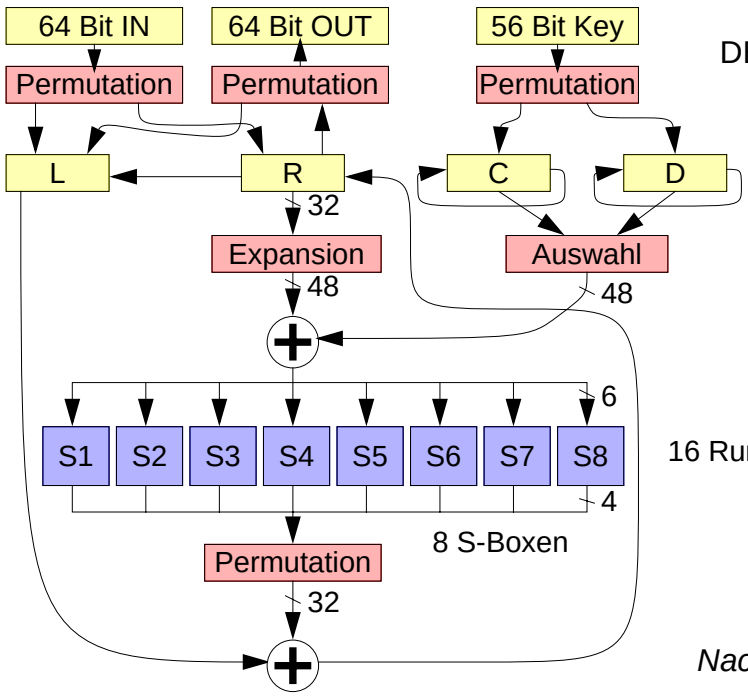




5.5 Heutige symmetrische Verfahren

- DES *(Data Encryption Standard, 1977)*
  - entwickelt von IBM
  - amerikanischer Standard (Kriegswaffe)
  - blockorientiertes Verfahren (64 Bit Block, 56 Bit Schlüssel)
  - 16 Runden
  - gilt heute als nicht mehr ganz sicher, da Rechenleistung von Großrechnern oder Rechenverbünden zum Brechen manchmal ausreicht

5.5 Heutige symmetrische Verfahren (2)



DES Blockdiagramm

16 Runden

Nach: Fumy/Rieß 1994

5.5 Heutige symmetrische Verfahren (3)

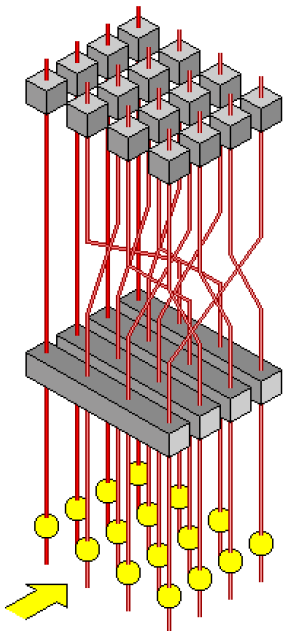
- Tripple DES
  - dreifache Verschlüsselung mit DES
  - Nutzung von drei oder mindestens zwei verschiedenen Schlüsseln
- IDEA *(International Data Encryption Algorithm)*
  - Alternative zu DES
  - 64 Bit Blockgröße
  - 128 Bit Schlüssel
  - keine Permutationen und S-Boxen
  - stattdessen: Addition, Multiplikation und XOR
  - 8 Runden und Output-Transformation
  - Einsatz: z.B. PGP *(Pretty Good Privacy)*

5.5 Heutige symmetrische Verfahren (4)

- AES — Advanced Encryption Standard (Rijndael)
  - entwickelt von Joan Daemen und Vincent Rijmen
  - blockorientiertes Verfahren
    - Blockgröße 128, 192 oder 256 Bits
    - Schlüsselgröße 128, 192 oder 256 Bits
  - 9, 11 oder 13 Runden je nach Schlüssellänge
  - wurde aus mehreren Vorschlägen als Nachfolger für DES ausgewählt

5.5 Heutige symmetrische Verfahren (5)

- Blockdiagramm einer Runde (stark vereinfacht)



Byte-Anordnung (hier 128 Bits)

Spalten-Anordnung

Spalten-Durchwürfelung

Schlüssel-Addition

Nach John Savard 2000

5.6 Beispiel: UNIX Passwörter

- Passwörter wurden zunächst im Klartext gespeichert
  - Passwortdatei muss streng geschützt werden
  - strenger Schutz oft nicht möglich (z.B. Backup der Platte)
  - Superuser kann die Passwörter von Benutzern einsehen
- Verschlüsseln der Passwörter
  - nur die verschlüsselte Version wird gespeichert
  - verschlüsselte Passwörter dürfen nicht leicht entschlüsselt werden können
- Ausprobieren von Passwörtern
  - Benutzer wählen Namen und Gegenstände als Passwort
  - Verschlüsseln von gängigen Begriffen und Vergleich mit verschlüsselt gespeicherten Passwörtern
  - Verschlüsselungszeit fließt mit ein in die Sicherheitsbetrachtung

5.6 Beispiel: UNIX Passwörter (2)

- Heutiges Verfahren
  - zufällige Auswahl eines von 4096 Werten *(Salt)*
  - der Salt fließt mit in die Verschlüsselung ein, so dass ein und dasselbe Passwort in 4096 Varianten vorkommen kann
  - Verschlüsselung mit DES
  - Zugriff auf verschlüsselte Passwörter wird weitestgehend verhindert (Shadow-Passwortdatei)
- ★ Vorteil
  - Ausprobieren von Passwörtern benötigt mehr Zeit
  - Vergleich zweier Passwörter weitgehend unmöglich

5.6 Beispiel: UNIX Passwörter (3)

- Politik am Institut für Informatik
  - Mindestlänge 8 Zeichen
  - mindestens 5 verschiedene Zeichen
  - mindestens 3 Zeichenklassen (Groß-, Kleinbuchstaben, Ziffern, Sonderzeichen)
  - keine Wiederholungen von Zeichenfolgen erlaubt
  - keine aufeinanderfolgenden Zeichen erlaubt, z.B. "123"
  - ...
  - Begriffe, Namen, etc. werden ausgeschlossen und müssen hinreichend verfremdet sein
- ★ Angriff durch Ausprobieren wird weitestmöglich erschwert

5.7 Heutige asymmetrische Verfahren

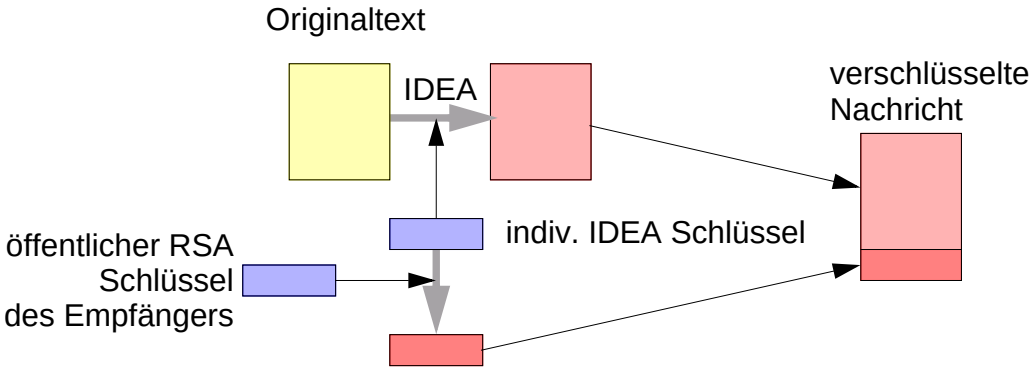
- RSA (Rivest, Shamir und Adleman)
  - Öffentlicher Schlüssel (zum Verschlüsseln) besteht aus *(e, N)*
  - Ein Block *M* wird verschlüsselt durch: $C = E((e, N), M) = M^e \bmod N$
  - *C* wird entschlüsselt durch: $M = D((d, N), C) = C^d \bmod N$
  - Wahl der Schlüssel:
    - Es muss gelten $\forall M : (M^e)^d = M \bmod N$
    - Aus Kenntnis von *e* und *N* darf *d* nur mit hohem Aufwand ermittelbar sein
  - Lösung:
    - $N = pq$ mit *p* und *q* zwei hinreichend große Primzahlen
    - zufällige Wahl von *d*, teilerfremd zu $(p-1)(q-1)$
    - Berechnung von *e* aus der Bedingung: $ed = 1 \bmod ((p-1)(q-1))$
  - Es ist aufwendig, die Primfaktoren von *N* zu berechnen (mit diesen wäre es möglich *d* zu ermitteln)

5.7 Heutige asymmetrische Verfahren (2)

- Vorteil asymmetrischer Verfahren (*Public key*-Verfahren)
  - nur ein Schlüsselpaar pro Teilnehmer nötig
    (sonst ein Schlüsselpaar pro Kommunikationskanal!)
  - Schlüsselverwaltung erheblich vereinfacht
    - jeder Teilnehmer erzeugt sein Schlüsselpaar und
    - veröffentlicht seinen öffentlichen Schlüssel
  - Authentisierung durch digitale Unterschriften möglich
  - gilt als sicher bei hinreichend großer Schlüssellänge (1024 Bit)
- Nachteil
  - relativ langsam berechenbar
  - gemischter Betrieb von asymmetrischen und symmetrischen Verfahren zur Geschwindigkeitssteigerung

5.7 Heutige asymmetrische Verfahren (3)

- Beispiel: PGP Verschlüsselung



  - Daten werden mit einem individuellen Schlüssel IDEA-verschlüsselt
  - IDEA-Schlüssel wird RSA-verschlüsselt der Nachricht angehängt

5.7 Heutige asymmetrische Verfahren (4)

- ★ Nachricht an mehrere Adressaten verschickbar
  - lediglich der IDEA-Schlüssel muss in mehreren Varianten verschickt werden
    (je eine Version verschlüsselt mit dem öffentl. Schlüssel des jeweiligen Empfängers)

5.8 Digitale Unterschriften

- Authentisierung des Absenders
  - Bilden eines Hash-Wertes über die zu übermittelnde Nachricht
    - Hash-Wert ist ein Codewort fester Länge
    - es ist unmöglich oder nur mit hohem Aufwand möglich, für einen gegebenen Hash-Wert eine zugehörige Nachricht zu finden
  - Verschlüsseln des Hash-Wertes mit dem geheimen Schlüssel des Absenders (digitale Unterschrift, digitale Signatur)
  - Anhängen des verschlüsselten Hash-Wertes an die Nachricht
  - Empfänger kann den Hash-Wert mit dem öffentlichen Schlüssel des Absenders dechiffrieren und mit einem selbst berechneten Hash-Wert der Nachricht vergleichen
  - stimmen beide Werte überein muss die Nachricht vom Absender stammen, denn nur der besitzt den geheimen Schlüssel

5.8 Digitale Unterschriften (2)

- Kombination mit Verschlüsselung
  - erst signieren
  - dann mit dem öffentlichen Schlüssel des Adressaten verschlüsseln
  - sonst Signatur verfälschbar
- Reihenfolge wichtig
  - Man signiere nichts, was man nicht entschlüsseln kann / versteht.
- Heute gängiges Hash-Verfahren
  - MD5
  - 128 Bit langer Hash-Wert

5.8 Digitale Unterschriften (3)

- Woher weiß ich, dass ein öffentlicher Schlüssel authentisch ist?
  - Ich bekomme den Schlüssel vom Eigentümer (persönlich, telefonisch).
    - Hash-Wert auf öffentlichen Schlüsseln, die leichter zu überprüfen sind (Finger-Prints)
  - Ich vertraue jemandem (Bürge), der zusichert, dass der Schlüssel authentisch ist.
    - Schlüssel werden von dem Bürgen signiert.
    - Bürge kann auch eine ausgezeichnete Zertifizierungsstelle sein.
    - Netzwerk von Zusicherungen auf öffentliche Schlüssel *(Web of Trust)*
  - Möglichst weite Verbreitung von öffentlichen Schlüsseln ereichen (z.B. PGP: Webserver als Schlüsselserver)

5.8 Digitale Unterschriften (4)

- Mögliche Probleme von Public key-Verfahren
  - Geheimhaltung des geheimen Schlüssels (Time sharing-System, Backup; Schlüsselpasswort / Pass phrase)
  - Vertrauen in die Programme (z.B. PGP)
  - Ausspähung während des Ver- und Entschlüsselungsvorgangs

6 Authentisierung im Netzwerk

- Viele Klienten, die viele Dienste in Anspruch nehmen wollen
  - Dienste *(Server)* wollen wissen welcher Benutzer *(Principal)*, den Dienst in Anspruch nehmen will (z.B. zum Accounting, Zugriffsschutz, etc.)
  - Im lokalen System reicht die (durch das Betriebssystem) geschützte Benutzerkennung (z.B. UNIX UID) als Ausweis
  - Im Netzwerk können Pakete abgefangen, verfälscht und gefälscht werden (einfache Übertragung einer Benutzerkennung nicht ausreichend sicher)
- Public key-Verfahren
  - Authentisierung durch digitale Unterschrift (mit geheimen Schlüssel des Senders) und Verschlüsseln (mit öffentlichem Schlüssel des Empfängers)
  - Nachteile
    - jeder Dienst benötigt sicheren Zugang zu allen öffentlichen Schlüsseln
    - Verschlüsseln und Signieren mit RSA ist sehr teuer

6 Authentisierung im Netzwerk (2)

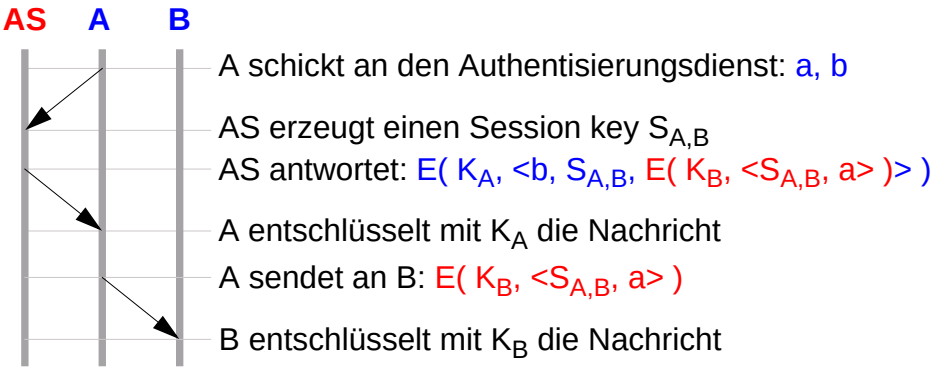
- ★ Einsatz von Authentisierungsdiensten
  - zentraler Server, der alle Benutzer kennt
  - Authentisierungsdienst garantiert einem Netzwerkdienst, dass ein Benutzer auch der ist, der er vorgibt zu sein
- Benutzerausweis
  - Authentisierungsdienst erkennt den Benutzer anhand eines geheimen Schlüssels oder Passworts
  - Schlüssel ist nur dem Authentisierungsdienst und dem Benutzer bekannt

6 Authentisierung im Netzwerk (3)

- Vorgang
  - Benutzer *(Principal)* will mit einem Programm *(Client)* einen Dienst *(Server)* in Anspruch nehmen
  - durch geeignetes Protokoll erhalten Client und Server jeweils einen nur ihnen bekannten Schlüssel, mit dem sie ihre Kommunikation verschlüsseln können *(Session key)*

6.1 Einfacher Authentisierungsdienst

- A will den Dienst B in Anspruch nehmen (Nach Needham-Schröder):



  - K_X ist der geheime Schlüssel, den nur Authentisierungsdienst und X kennen
  - nach dem Protokollablauf kennen sowohl A und B den Session key
  - A weiß, dass nur B den Session key kennt
  - B weiß, dass nur A den Session key kennt

6.1 Einfacher Authentisierungsdienst (2)

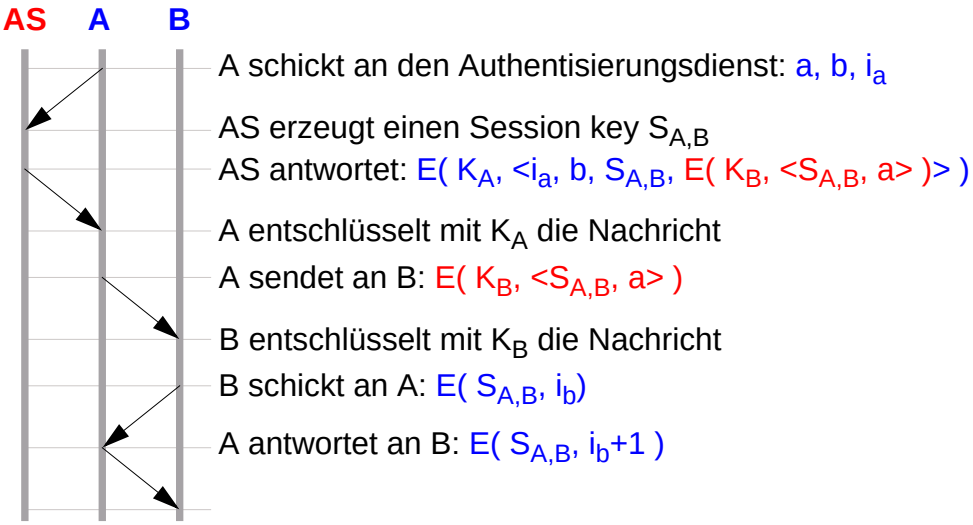
▲ Problem

- ◆ letzte Nachricht von A an B könnte aufgefangen und später erneut ins Netz gegeben werden *(Replay attack)*
- ◆ Folge: Kommunikation zwischen A und B kann gestört werden

★ Korrektur durch zusätzliches Versenden einer Verbindungsbestätigung durch B und A

6.2 Authentisierungsdienst mit Bestätigung

■ Bestätigung enthält Einmalinformation *(Nonce)*



- ◆ ein Wiedereinspielen der Nachricht E(K_B, <$S_{A,B}$, a>) oder E($S_{A,B}$, i_b+1) wird erkannt und kann ignoriert werden

6.2 Authentisierungsdienst mit Bestätigung (2)

▲ Problem

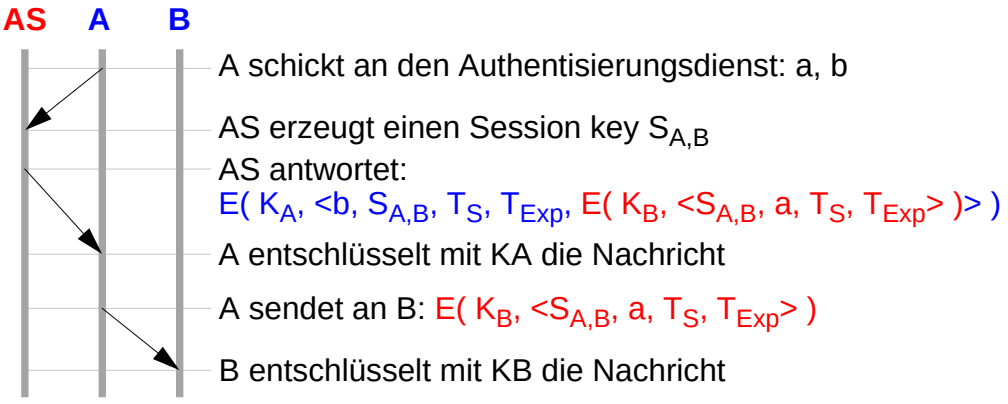
- ◆ Aufzeichnen von $E(K_B, <S_{A,B}, a>)$ und
- ◆ Brechen von $S_{A,B}$ erlaubt das Aufbauen einer Verbindung.
- ◆ ein Dritter kann dann die erste Bestätigung abfangen und die zweite Bestätigung verschicken

★ Lösung

- ◆ Einführung von Zeitstempeln *(Time stamp)* und Angaben zur Lebensdauer *(Expiration time)*

6.3 Authentisierungsdienst mit Zeitstempeln

■ Authentisierungsdienst versieht seine Nachrichten mit Zeitstempeln



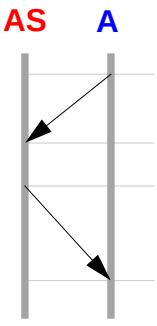
- ◆ T_S = Zeitstempel der Nachrichtenerzeugung
- ◆ T_{Exp} = maximale Lebensdauer der Nachricht
- ◆ aufgezeichnete Nachricht kann nach kurzer Zeit (z.B. 5min) nicht nocheinmal zum Aufbau einer Verbindung verwendet werden

6.4 Beispiel: Kerberos

- Kerberos V5
  - Softwaresystem implementiert Weiterentwicklung des Needham-Schröder-Protokolls
  - entwickelt am MIT seit 1986
- Ziel
  - Authentisierung und Erzeugung eines gemeinsamen Schlüssels durch den vertrauenswürdigen Kerberos-Server
- Idee
  - Trennung von Authentisierungsdienst und Schlüsselerzeugung
  - reduziert die nötige Übertragung einer Identifikation oder eines Passworts zum Kerberos-Server

6.4 Beispiel: Kerberos (2)

- Benutzer holt sich zunächst ein Ticket vom Authentisierungsdienst



A schickt an den Authentisierungsdienst: a, tgs, T_{Exp}, i_a

AS erzeugt ein Ticket für den Ticket Server tgs

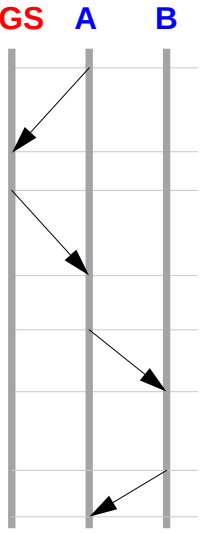
AS antwortet:
E(K_A, <$S_{A,TGS}$, tgs, T_{Exp}, i_a, E(K_{TGS}, <$S_{A,TGS}$, a, T_{Exp}>)>)

A entschlüsselt mit KA die Nachricht

- das Ticket besteht aus <$S_{A,TGS}$, a, T_{Exp}>
- es enthält einen Session key für die Kommunikation mit einem Ticket granting server, der dann die Verbindung zu einem Netzwerkdienst bereitstellen kann

6.4 Beispiel: Kerberos (3)

- Authentisierungsdienst versieht seine Nachrichten mit Zeitstempeln



A schickt an den Ticket granting server:
$E(S_{A,TGS}, T_s), E(K_{TGS}, <S_{A,TGS}, a, T_{Exp}>), b, T_{Exp}, i_a'$

TGS erzeugt einen Session key $S_{A,B}$

TGS antwortet:
$E(S_{A,TGS}, <S_{A,B}, b, T_{Exp}, i_a'>), E(K_B, <S_{A,B}, a, T_{Exp}>)$

A entschlüsselt mit $S_{A,TGS}$ die Nachricht

A sendet an B: $E(S_{A,B}, <T_s, C, S_{sub}>), E(K_B, <S_{A,B}, a, T_{Exp}>)$

B entschlüsselt mit K_B den zweiten Teil der Nachricht (Ticket) und mit $S_{A,B}$ den ersten Teil

B antwortet (optional): $E(S_{A,B}, T_s)$

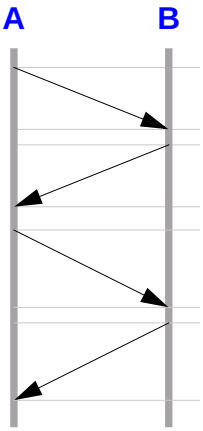
  - C = Checksumme zur Überprüfung der richtigen Entschlüsselung
  - A kann mehrere Verbindungen mit seinem Ticket öffnen

6.4 Beispiel: Kerberos (4)

- Unterscheidung zwischen Benutzer *(User)* und Benutzerprogramm *(Client)*
  - Wie kann ein Benutzerprogramm seinen Benutzer identifizieren?
  - Geheimer Schlüssel vom Benutzer (K_X) hängt von einem Passwort ab
  - mittels einer Einwegfunktion wird aus dem Passwort der Schlüssel K_X erzeugt
  - Benutzerprogramm braucht also das Passwort zur Verbindungsaufnahme
- Beispiel: *kinit, klogin*
  - Anmeldung beim Authentisierungsdienst mit *kinit* und Passworteingabe
  - Ticket wird im Benutzerverzeichnis gespeichert
  - *klogin* erlaubt das Einloggen auf einem entfernten Rechner mit Datenverschlüsselung und ohne Passwort

6.5 Austausch öffentlicher Schlüssel

- A und B tauschen ihre öffentlichen Schlüssel aus



A schickt an B seinen öffentlichen Schlüssel: K_A

B schickt an A seinen öffentlichen Schlüssel: K_B

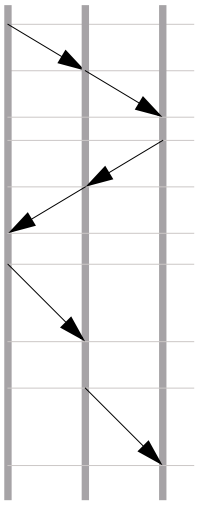
A verschickt signierte und verschlüsselte Nachricht:
$E(K_B, E(K_A^{-1}, T_1))$

B verschickt signierte und verschlüsselte Nachricht:
$E(K_A, E(K_B^{-1}, T_2))$

- Problem
  - A und B können nicht sicher sein, dass der öffentliche Schlüssel wirklich vom jeweils anderen stammt

6.5 Austausch öffentlicher Schlüssel (2)

- Aktiver Mithörer C fängt Datenverbindungen ab *(Man in the middle attack)*



A versucht an B seinen öffentlichen Schlüssel zu schicken: K_A

C fängt Nachricht ab und schickt an B seinen eigenen öffentlichen Schlüssel: K_C

B versucht an A seinen öffentlichen Schlüssel zu schicken: K_B

C fängt Nachricht ab und schickt an A seinen eigenen öffentlichen Schlüssel: K_C

A versucht an B eine Nachricht zu schicken:
$C_1 = E(K_C, E(K_A^{-1}, T_1))$

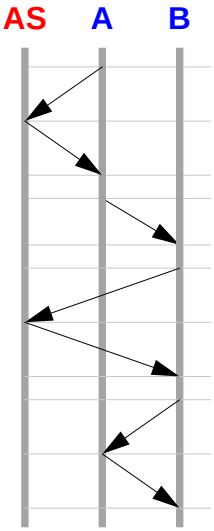
C dekodiert die Nachricht: $T_1 = D(K_C^{-1}, D(K_A, C_1))$

C schickt die Nachricht weiter: $E(K_B, E(K_C^{-1}, T_1))$

B glaubt eine Nachricht von A zu empfangen

6.5 Austausch öffentlicher Schlüssel (3)

- Einsatz eines Authentisierungsdienstes



A kontaktet AS und möchte Verbindung mit B: a, b

AS verschickt den Schlüssel von B: $E(K_{AS}^{-1}, <K_B, b>)$

A schickt initiale Nachricht an B: $E(K_B, <a, i_a>)$

B kontaktet AS und möchte den Schlüssel von A: b, a

AS verschickt den Schlüssel von A: $E(K_{AS}^{-1}, <K_A, a>)$

B antwortet an A: $E(K_A, <i_a, i_b>)$

A bestätigt: $E(K_B, <i_b>)$

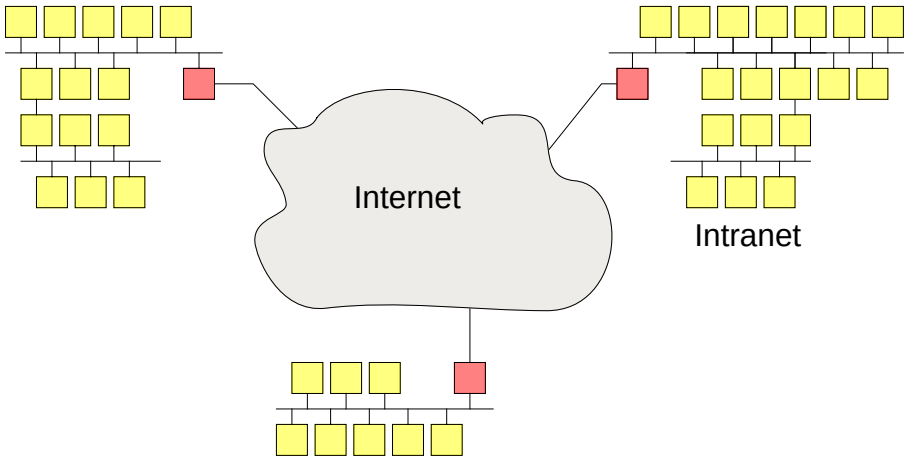
- Replay-Probleme
  - Hinzunahme von Zeitstempel und Lebendauer

7 Firewall

- Trennung von vertrauenswürdigen und nicht vertrauenswürdigen Netzwerksegmenten durch spezielle Hardware *(Firewall)*
  - Beispiel: Trennen des firmeninternen Netzwerks (Intranet) vom allgemeinen Internet
- Funktionalität
  - Einschränkung von Diensten
    - von innen nach außen, z.B. nur Standarddienste
    - von außen nach innen, z.B. kein Telnet, nur WWW
  - Paketfilter
    - Filtern „defekter“ Pakete, z.B. SYN-Pakete
  - Inhaltsfilter
    - Filtern von Pornomaterial aus dem WWW oder News
  - Authentisieren von Benutzern vor der Nutzung von Diensten

7 Firewall (2)

- Virtual private network
  - Verbinden von Intranet-Inseln durch spezielle Tunnels zwischen Firewalls
  - getunnelter Datenverkehr wird verschlüsselt
  - Benutzer sieht ein „großes Intranet“ (nur virtuell vorhanden)



8 Richtlinien für den Benutzer

8.1 Passwörter

- Wahl eines Passworts
  - hinreichend komplexe Passwörter wählen
  - Schutz vor Wörterbuchangriffen
  - verschiedene Passwörter für verschiedene Aufgaben
    (z.B. PPP-Passwort ungleich Benutzerpasswort)
- Aufbewahrung
  - möglichst nirgends aufschreiben
  - nicht weitergeben
  - kein Abspeichern auf einem Windows-Rechner (Option immer wegklicken)

8.1 Passwörter

- Eingabe
  - niemals über eine unsichere Rechnerverbindung eingeben
    - **`ftp, telnet, rlogin`** Dienste vermeiden
    - nur sichere Dienste verwenden: **`ssh, slogin`**
    - Datenweg beachten, über den das Passwort läuft: ein unsicheres Netzwerk ist bereits genug
- Änderung
  - Passwörter regelmäßig wechseln
  - alte Passwörter nicht wiederverwenden

8.2 Schlüsselhandhabung

- Einsatz von PGP oder S/MIME
  - Zugang zu den privaten Schlüsseln für andere verhindern
  - Dateirechte auf der Schlüsseldatei prüfen
  - privater Schlüssel nur auf Diskette
  - Passphrase wie ein Passwort behandeln
  - privaten Schlüssel nie über unsichere Netze transportieren

8.3 E-Mail

- Authentisierung
  - Bei elektronischer Post ist der Absender nicht authentisierbar
  - Digitale Unterschriften einsetzen (z.B. mit PGP oder S/MIME)
- Abhören
  - Elektronische Post durchläuft viele Zwischenstationen und kann dort jeweils gelesen und verfälscht werden
  - Verschlüsselung einsetzen (z.B. mit PGP oder S/MIME)

8.4 Programmierung

- S-Bit Programme vermeiden
  - Oft kann das S-Bit durch geschickte Vergabe von Benutzergruppen an Dateien vermieden werden
- Verwendung zusätzlicher Rechte (z.B. durch S-Bit) nur in Abschnitten
  - Trusted Computing Base (TCB) [hier kein vollständiger Schutz]

```
seteuid(getuid());/* am Programmanfang Rechte wegnehmen */
...
seteuid(0);         /* setzt root Rechte */
fd = open("/etc/passwd", O_RDWR);
seteuid(getuid());/* nimmt root Rechte wieder weg */
...
```

- Sorgfältige Programmierung
  - Funktionen wie **`strcpy, strcat, gets, sprintf, scanf, sscanf, system, popen`** vermeiden oder durch **`strncpy, fgets, snprintf`** ersetzen

8.5 World Wide Web

- Cookies
  - Akzeptieren von Cookies erlaubt einer Website die angesprochenen Seiten genau einem Benutzer zuzuordnen
  - funktioniert über Sessions hinweg
- JavaScript
  - schwere Sicherheitslücken erlauben es, alle für den Benutzer lesbare Dateien an einen Dritten weiterzugeben
  - Ausschalten!
- Abhören
  - WWW-Verbindungen können abgehört werden
  - keine privaten Daten, wie z.B. Kreditkartennummern übertragen
  - Secure-HTTP mit SSL-Verschlüsselung benutzen; https-URLs